12-2017

The Design of a Custom 32-Bit SIMD Enhanced Digital Signal Processor

Shashank Simha
ss7841@rit.edu

The Design of a Custom 32-bit SIMD Enhanced Digital Signal Processor

by
Shashank Simha

Graduate Paper

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in Electrical Engineering

Approved by:

Mr. Mark A. Indovina, Lecturer
Graduate Research Advisor, Department of Electrical and Microelectronic Engineering

---

Dr. Sohail A. Dianat, Professor
Department Head, Department of Electrical and Microelectronic Engineering

Department of Electrical and Microelectronic Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
December 2017

To my family and friends, for all of their endless love, support, and encouragement throughout my career at Rochester Institute of Technology

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this paper are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This paper is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Shashank Simha

December 2017

Acknowledgements

I would like to thank my advisor, professor, and mentor, Mark A. Indovina, for all of his guidance throughout the entirety of this project. The continuous feedback and motivation provided by him has been a major driving force to push myself beyond limits throughout my career at RIT, for which I am truly grateful. His passion for teaching, expertise in digital design, along with decades of industrial experience has established him as my role model in the field. His advice, methods of teaching, managing and cross-domain knowledge has been a huge inspiration for me to pursue a career in the VLSI and digital design.

I would like to thank Dr. Dorin Patru and Dr. Marcin Lukowiak for providing me valuable knowledge and feedback in topics of computer architecture and FPGA, which provided a firm foundation in my understanding of the topics.

I would like to thank my parents for their continuous support throughout my career at RIT, believing in me and my being biggest role models. They have always been my pillars of support and great motivators throughout my life, at and away from home.

I would also like to thank my roommates for being my brothers throughout the two years of graduate school.

I finally would like to thank all my classmates and TA's for their invaluable guidance and support throughout my entire career at RIT.

Abstract

For a number of years, the hardware industry has seen a drastic rise in embedded applications. Thanks to the Internet of Things (IoT) revolution, a majority of these embedded applications are shifting towards the usage of simple hardware capable of running on batteries, while being able to handle complex data and implement complex algorithms. Translating these requirements to digital design terms, the hardware is expected to have high power efficiency, be tiny and simple enough, while being capable of meeting real-time constraints and process mathematical algorithms. Looking at some of the modern DSPs, most of them have been targeting high performance and wider applications, usually resulting in higher power consumption and complex hardware.

The main motivation of this paper was to implement a simple DSP design, optimized for power efficiency, while being capable of handling simple multimedia applications. Hence, an enhanced version of TMS32010 DSP is implemented with numerous modifications to the architecture, ISA, memory addressing and pipeline structure. The major enhancements include the addition of instruction level parallelism using SIMD instructions, use of a much larger data memory to be able to accommodate a larger amount of data in multimedia applications, and expansion of the data-word to 32-bits to be able support packed SIMD data and fully utilize the 32-bit ALU. The ISA, pipeline and memory access enhancements target higher power efficiency by using a single clock across the design.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With advancement in technology, the world has been seeing exponential increase in the amount of data stored and processed ever since computers have been invented. A major part of this data represents multimedia, which is essentially either audio or image data [1]. To clearly compress, restore, process and understand image data, numerous mathematical algorithms have been implemented in computing, which are usually quite complex. After the invention of general purpose processors, there were many applications where a lot of its functions were not required by the application, or used by limited applications [2]. And, these processors took too much time to compute the mathematically intense algorithms in real time, which the hardware was simply not built to handle. This market was targeted by DSPs (Digital Signal Processors). DSPs have historically been used in such applications to increase the speed of computing by implementing complex hardware and parallel computing [3].

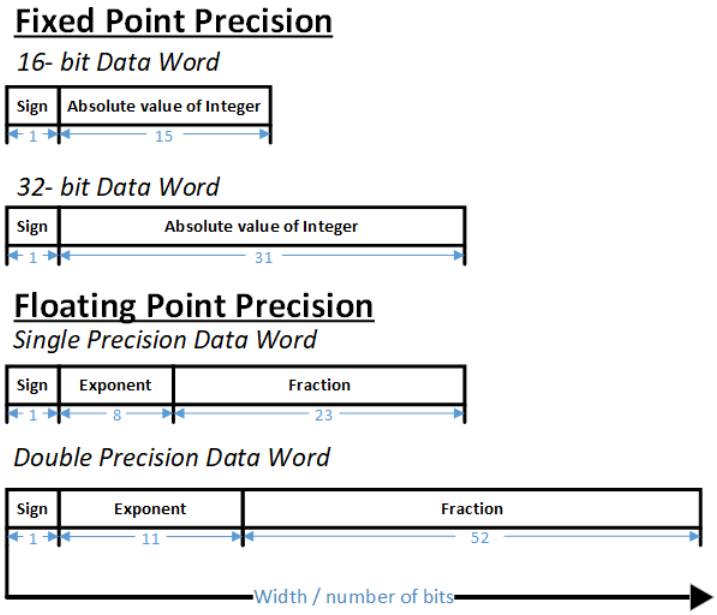

Figure 1.1: Fixed and floating point illustration

1.1 DSP classifications

DSPs are broadly classified into fixed and floating-point architectures. Fixed-point DSPs are designed to handle positive or negative integer data, while floating-point DSPs are designed to handle rational number data. The representation of data stored in each of these DSPs hence is different, which is the major reason behind the classification since it directly affects the amount of hardware required for each implementation. The fixed-point data is represented by the integer's sign in the MSB (Most Significant Bit) followed by its value in the following bits. Floating-point data is represented by the rational number's sign in the MSB, followed by its exponent, and later its mantissa. Fig. 1.1 illustrates fixed and floating point representations.[4]

Generally, fixed point implementations are faster, cheaper, more power efficient, simpler to design and verify, and require less time-to-market. Floating point implementation

trades-off all these factors for better precision and faster computation of floating point data. Hence most of the times, either of the architectures is selected mainly based on the application. It is also worth noting that some DSPs are equally efficient in implementing either architectures like the SHARC DSP by Analog Devices.

Architecturally, the amount of hardware and design effort required to implement floating point precision is obviously much higher than fixed point precision. First, the data unit must be expanded from 16-bits to 32-bits at least along with the memory and registers. The ISA itself would have to be expanded significantly as almost all floating point DSPs usually support fixed point operations along with floating point ones [5].

From a compilation point of view, C language has an in-built type for floating-point to fully exploit the hardware capability of floating point DSPs. While the C compiler takes advantage of the floating point hardware, some rules and regulations are followed to ensure that the data fits within the 32-bit or 64-bit data word. Fixed point C compilation is implemented by mapping integers to fixed point data. The problem with fixed point though is that there is no ANSI standard for fixed point, hence it usually requires additional code for conversions and shifts. The efficiency of fixed point compilation takes another dip, since fixed point specific instructions are not built-in [6].

1.2 History of DSPs

The main motivation for DSP was to have powerful hardware with more application specific functions and instructions, when compared to a general-purpose processor. This is very evident throughout the evolution of DSPs looking at the various applications throughout the past few decades. Moreover, it is obvious that all early DSPs were fixed-point architectures mainly because there was no floating-point standardization until early 80's. With

applications ranging from audio systems, speech processing, SONAR to medical imaging, RADAR, DSPs are used almost in every field today [7]. It is also interesting to note early applications of DSPs in personal computers like the Motorola 56000 used in the Atari Falcon, NeXT and SGI workstations.

DSPs have been produced by almost all major semiconductor companies including Intel, AMD, Texas Instruments, Motorola and Analog Devices at some point of time. Most of the early DSPs targeted audio processing, such as the Speak & Spell by Texas Instruments. Throughout the evolution of DSPs, they have grown more and more application specific over time, rather than the other way around [5].

Speak & Spell, an early toy used to teach kids to spell words, launched in 1976, was the earliest mass-produced DSP product in the market, powered by the Texas Instrument TMS5100 DSP [8]. Interestingly, in the late 70's Intel unsuccessfully tried to enter the DSP market early with their 2920 analog processor, which failed mainly because of the absence of a true multiplier. The first attempts of DSP devices include the AT&T DSP1 and NEC µPD7720. It is worth noting that DSP1 introduced the historic MAC instruction to the world, this was one of the earliest steps of implementing instruction level parallelism in DSPs [9].

The first generation of DSPs started appearing in the market in the early 1980's. Some key features of the DSPs of this generation were Harvard architecture and multiply-add-accumulate instructions. The TMS32010, from this generation of DSPs, by Texas Instruments was notably one of the most successful DSPs in history, as it pushed Texas Instruments to be the market leader in DSPs. Since it was based on Harvard architecture and with specialized ISA, it was the fastest DSP at the time [9].

The next generation DSPs, from late 1980's to early 1990's featured advanced architectures with capability of handling much complex applications. Motorola entered the

DSP market with their popular fixed-point DSP56000 featuring 24-bit program and data words. The second generations of DSPs featured further optimization in memory architecture, with architectures capable of accessing multiple data memories in a single instruction. This generation also brought floating-point DSP architecture into market. Examples for this include the SHARC series of DSPs by Analog Devices, calling the architecture Super Harvard architecture. Interestingly, shrinking fabrication technology also had a huge impact on this generation of DSPs, as more and more hardware could be fit into the chip while still keeping it tiny in size [9].

Late 90's DSPs incorporated more application-specific instructions, as they were mostly used as coprocessors along with the main CPU. Many DSPs however lost market when CPUs became SIMD capable. Parallel processing capabilities were subsequently introduced with Single Instruction Multiple Data (SIMD) and Very Long Instruction Word (VLIW) instructions in the later DSPs. VLIW architectures take advantage of spatial parallelism along with temporal parallelism, since they utilize several functional units to concurrently execute multiple operations, while pipelining these functional units [10]. Parallelism was further boosted with adding multiple cores and threads in later DSPs [9].

Modern DSPs are functionally not very different from the late 90's ones. Optimizations in the last decade though has been directed toward DSP compilation strategies. Designing and making DSP architecture more compiler friendly and making better DSP compilers has been a very crucial step in DSP evolution, mainly because its applications will tremendously increase since it is the bridge to the software world. As the software world starts utilizing DSPs effectively and understanding their capabilities, significant advantages in software productivity could be achieved [11]. Also, modern DSPs have been trying to incorporate as much parallelism as possible, with different approaches. The latest Texas Instruments DSP TMS320C64X is very good example of this, as it combines both VLIW and SIMD

capabilities into the architecture. DSPs from LSI Logic Corp. take a different approach with their super-scalar architecture, while arguing that superscalar architecture is more compiler friendly compared to VLIW approach [9].

1.3 Brief introduction to the DSP design and paper organization

The capabilities of DSPs have evidently evolved at a rapid pace over time, especially since the 90's. With the introduction of concepts such as super Harvard architecure, VLIW and super-scalar architecture, the design complexity of DSPs has also risen at the same pace. Hence, the need for a simple embedded programmable processor with not only conventional instructions, but also DSP specific becomes desirable in some applications [12][13][14][15]. The paper [12] shows one such approach where, the TMS32010, a fairly simple monolithic DSP, is implemented on an FPGA platform.

The DSP design presented is very similar to the TMS32010 by Texas Instruments, but has major enhancements which are discussed in the later chapters. Looking at some applications of TMS32010 was a necessary step while deciding what enhancements could positively impact the DSP applications. Numerous applications of TMS32010 in the area of image processing involve design of a multiprocessor system using multiple DSPs to implement a complex algorithm. Paper [16] presents one such application where edge detection algorithm is implemented using eight TMS32010 DSPs in a multiprocessor configuration, involving parallel image processing architecture. Another interesting image processing application is presented in [17], where the TMS32010 is interfaced with a host processor and to speed up image processing algorithms. The paper also mentions limitations of the system, one of which is the lack of data memory in the TMS32010. This has been one of the

major enhancements in our DSP design.

The following chapters of the paper attempt to explain all details involved in designing the DSP. Chapter 2 talks about the architecture of the DSP, including the data flow within the DSP, types of operations and compares it other DSP architectures. Chapter 3 covers topics like instruction and data word expansions, opcodes of all instructions and addressing modes and assembler design, in an effort to describe the Instruction Set Architecture (ISA) of the DSP. Chapter 4 talks about pipeline design and RAM buffer wrapper design. Chapter 5 elaborates one application of the DSP which is the median filter design, and describes the merits of the DSP with respect to its implementation. Chapters 6 and 7 present the results and conclusion of the paper.

Chapter 2

DSP architecture

The DSP architecture is very different from that of a general-purpose CPU as discussed in the previous chapter. One of the biggest bottlenecks in executing DSP algorithms is transferring information to and from memory [5]. Things like Harvard architecture, direct memory access (DMA), multiply-accumulate unit (MAC) and barrel shifting are some features which distinguish DSPs from general purpose processors. Some DSPs have general purpose registers, like the SHARC ADSP-2106x, and others are accumulator based, such as the TMS32010 [5].

As noted earlier, the primary advantage of the DSP is its speed. This means its architecture needs to be capable of performing complex mathematical calculations within a single clock cycle. There are different techniques to achieve this architecturally. Firstly, pipelining the architecture ensures that most instructions are executed within a single clock cycle, but have a latency between the input instruction and its output equal to the number of pipelined stages. While this approach is attractive, one thing to remember is that the same resource can not be used in multiple stages. The second solution is parallel execution of multiple tasks. The important point to remember here that these tasks should not have

dependencies, and off course can not use the same resources. DSP architecture is usually designed by combining both techniques to accomplish the speed.

The DSP's architecture being its most important feature, its significant difference of the from that of conventional microprocessors comes obvious. The basic capability of integrating a multiplier/ accumulator into its data-path has been proven to be revolutionary in computing multiple algorithms. Other factors such as preserving the precision of the product after multiplication, having shift capability while storing accumulator into the memory and handling overflow are crucial for DSP architecture, since most of its applications are usually complex arithmetic operations, requiring precise calculations [18].

While chapters 3 and 4 deal with how exactly each instruction is planned and executed, and pipeline design of the DSP respectively, this chapter starts by taking a brief look at the architecture of top-level design. Later in the chapter, all other lower level blocks of the DSP including functional blocks of the architecture are discussed in detail.

2.1 Top level block diagram

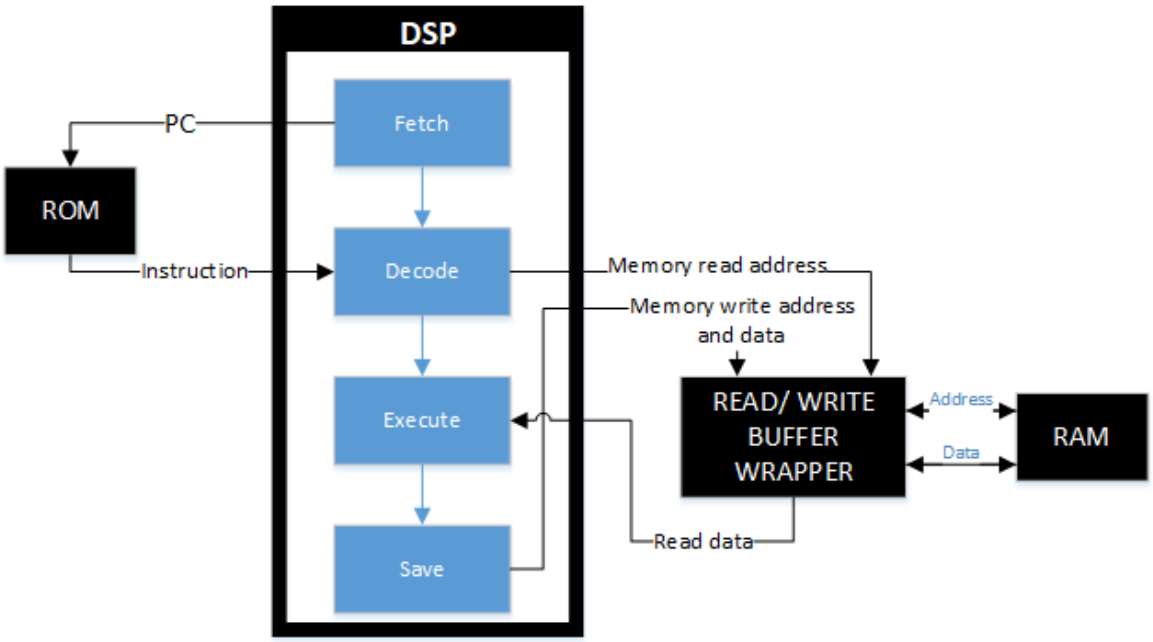

Figure 2.1: Top-level block diagram

The top-level block includes the DSP, ROM, RAM and read/write buffer wrapper. Fig. 2.1 attempts to visually describe the top-level view of the DSP system along with an abstracted high-level view of the interconnections between these components. It is to be noted that all the modules within the DSP are functional blocks, not pipeline stages. And, the necessity and usage of read-write RAM buffer is explained in chapter 4.

The DSP sends a new value for program counter (PC) every cycle, which goes to the ROM. The ROM returns the instruction corresponding to the previous PC, back to the DSP. Later, depending on the instruction, the DSP then sends a read address to the read/write RAM buffer to fetch operand/value from the RAM. The read/write RAM buffer in turn communicates this address to the RAM and accordingly fetches data from the RAM. This data is sent to the DSP, which executes the instruction and computes the

result. Lastly, depending on the following instruction, this result is saved into the RAM via the read/write RAM buffer.

2.2 Internal blocks

As discussed in the previous section, the DSP needs to perform the following functions for every instruction in the same order:

1. Fetch the instruction from ROM,
2. Decode this instruction,
3. Fetch data from RAM if necessary,
4. Execution the instruction, and
5. Save the result back into the RAM if required.

While the pipeline takes care of distributing these functions across multiple clock cycles, it is necessary to carefully plan the hardware necessary to perform each function.

The address decode unit 2.2.1 section describes how the instruction is broken into pieces as soon as the DSP receives it from the ROM. As the logic remains same for almost all instructions, most its implementation is described within the small subsection. However, execution being a more complex task, as it is unique for every instruction, requires further planning. Hence, the ALU 2.2.3 is separately discussed, following a brief look at the execution unit 2.2.2.

2.2.1 Address decode unit

Fig.2.2 shows the block diagram of decode unit. The decode unit supports two addressing modes, namely direct and indirect addressing modes. As clearly shown in the figure, eight Auxiliary Registers (ARs) are used for indirect addressing. The AR pointer (ARP) is used to indicate which AR is to be used. Chapter 3 discusses addressing modes in detail.

Direct addressing does not require much logic to decode, as the instruction itself contains almost all required details to generate the data memory address. Here, the least significant or LSB 7-bits of the instruction is simply concatenated with the contents of the data page pointer (DPP) to generate the data memory address.

While using indirect addressing, the instruction specifies the following details: which AR will be used for the next indirect addressing or the next AR pointer (NARP), and whether the contents of the current AR is to be incremented/decremented by one or not. The AR however contains the address of data to be fetched.

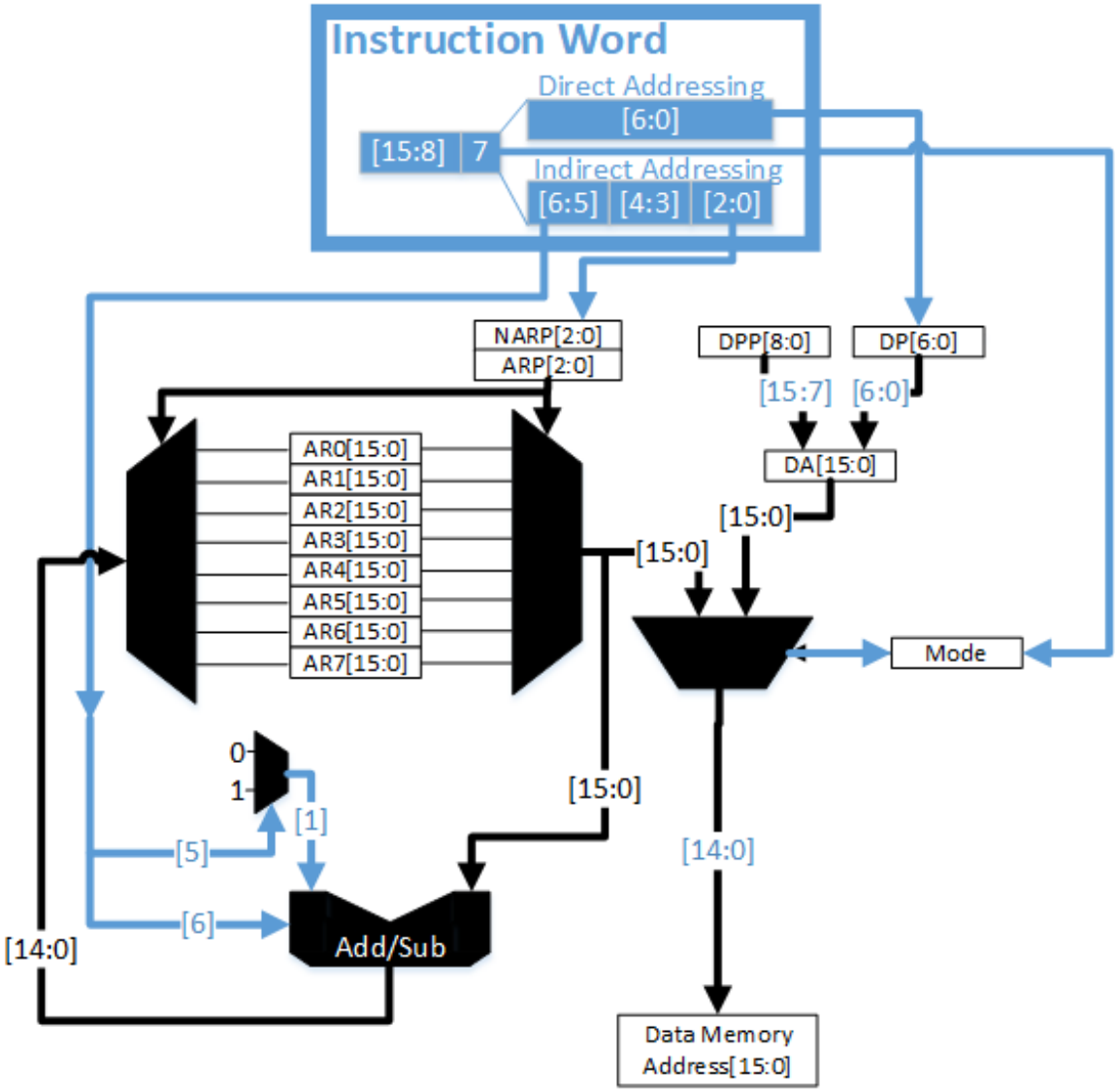

Figure 2.2: Address decode unit block diagram

2.2.2 Execution unit

Fig.2.3 describes the block diagram of the execution unit. The execution unit is broken down into three major functional blocks: ALU, barrel shifter and multiplier. Apart from these, all the other components in the figure are used to facilitate the interconnection

between these blocks, as directed by the instruction word decode logic.

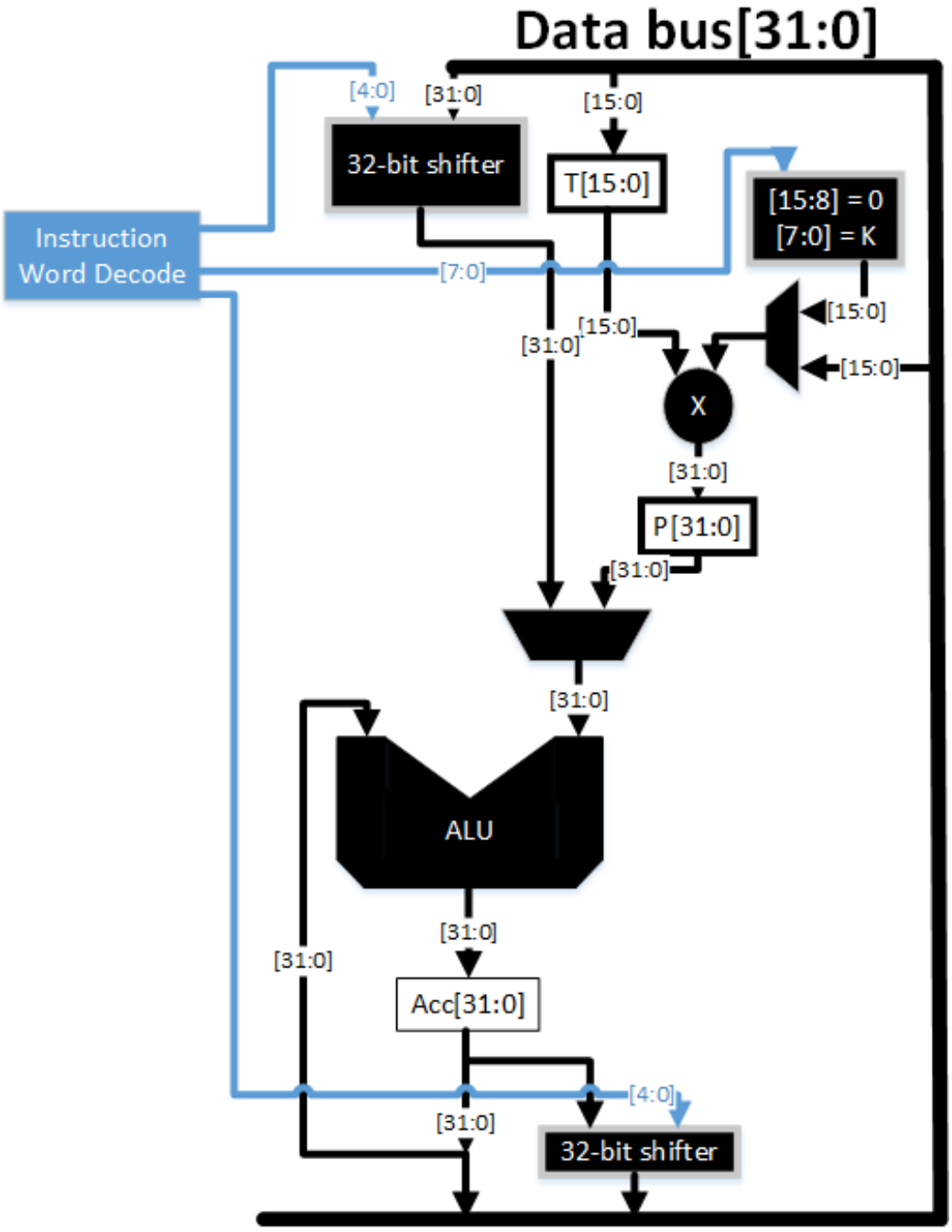

Figure 2.3: Execution unit block diagram

Two 32-bit barrel-shifters are used in the DSP. The input shifter is used to shift one of the ALU inputs, while the output shifter is used to shift the result of the ALU. The input data to the first shifter is read from the RAM. The second shifter is however used only while storing or writing back the accumulator contents into the RAM.

A 16x16 bit multiplier is used for multiplication operations. While one input to the multiplier always comes from the T-register, the other input can either be read from the RAM, or directly read from the instruction. The output however is always stored in the P-register.

The DSP is designed such that one of the ALU inputs is always the accumulator, while the other input is loaded either from the output of the first shifter or from the P-register. The result of the ALU is always fed back into the accumulator.

2.2.3 ALU

To accommodate SIMD instructions into the ISA, the ALU is optimized for add/subtract SIMD instructions. Fig. 2.4 shows the block diagram of the ALU. The main goal while designing the ALU was to add minimal hardware to TMS32010 design, while also supporting SIMD instructions.

Four sets of 8-bit adders were used to implement 32-bit addition-subtraction operations, as well as 8-bit SIMD addition-subtraction operations. The only change internally to the adders was the additional multiplexers between the carry signals of the consequent adders, which were set to zero for SIMD add. For the subtraction operation though, 32-bit 2's complement was fed to the adders in the case of 32-bit subtraction operation, while separate 2's complements were fed to the adders for each set of 8-bits in the case of SIMD subtractor.

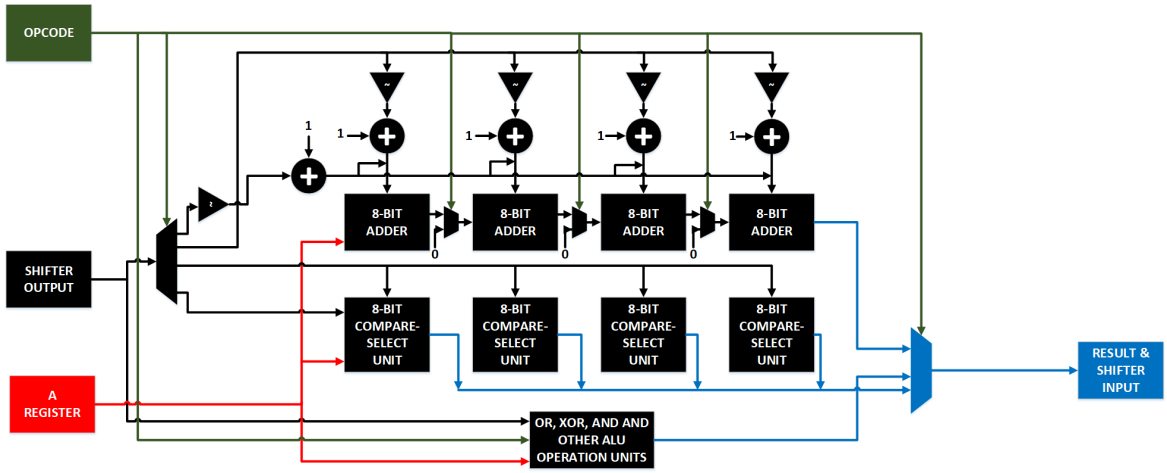

Figure 2.4: ALU block diagram

Chapter 3

Instruction Set Architecture of the DSP

One of the biggest challenges while designing a DSP is to strike a fair balance between the hardware complexity for implementing a particular ISA (Instruction Set Architecture), and possible applications of almost every instruction. For the past few decades, major DSP manufacturers have been experimenting with different instructions and ISAs to maximize the applications of their DSPs across various fields. While having a complex ISA and hundreds of instructions looks like a clear winner, a significantly huge number of DSPs are used in embedded and real-time applications where power, size and cost are extremely important. Interestingly, with the advent of Internet of things (IoT), there has been an exponential increase in such applications. Here, ISA complexity needs to be traded off for flexibility and robustness.

Another very important factor to consider is the type of addressing, or how easy or flexible the address conversion logic is for a user. This part is even more crucial in DSPs as almost all DSPs fetch data from the RAM for each ALU operation, unlike CPUs where

numerous general-purpose registers are used as ALU operands. And, it is quite obvious that most of the DSP operations would be ALU operations. [19]

The proposed DSP architecture is very similar to TMS32010 in its ISA. In the next few sections, the instruction and data words of the DSP, addressing modes and instruction opcodes along with operations are briefly described.

3.1 Instruction and data word expansion

To be able to fit in all the instructions and corresponding data required for each instruction within 16-bits of instruction word, five types of instruction words are planned. Fig. 3.1 shows the expansions of these instruction words. It is to be noted that the expansions indicated in the figure are only for direct addressing mode. In the case of indirect addressing, the last 7-bits are used differently. Bits 0, 1 and 2 are used to indicate the value of next AR pointer (NARP), while bits 5 and 6 are used to indicate post increment/decrement operation for the current AR.

Data words can be stored in four different formats, depending on the type of instruction used to handle them. Fig. 3.2 shows all variants of data word, where D0, D1, D2 and D3 are 8-bit signed/unsigned integers. While all instructions handled by the ALU are 32-bit data words, SIMD instructions use the 8-bit variants.

3.2 Addressing modes

As indicated while describing the decode unit in chapter 2, two addressing modes are implemented in the DSP. These addressing modes function exactly like the TMS32010, except eight Auxiliary registers are used here instead of two. Also, since the DSP design

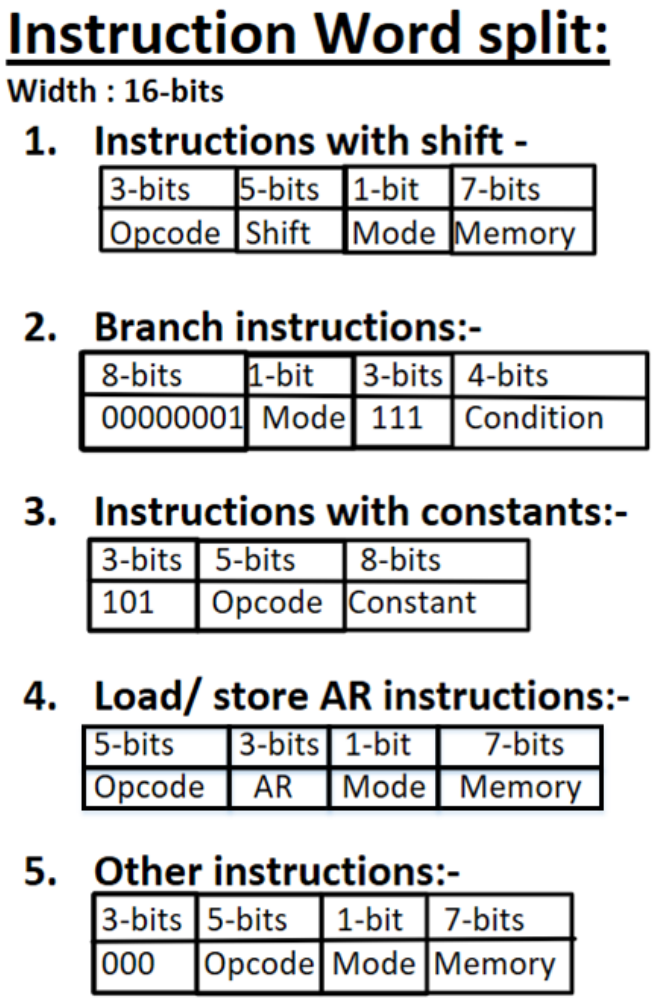

Figure 3.1: Instruction word expansion for various instructions

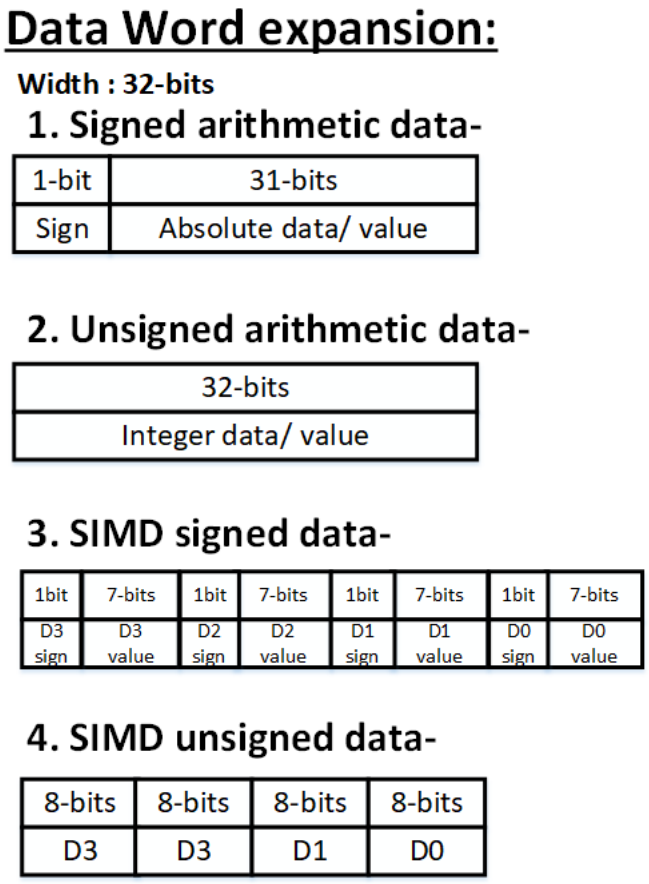

Figure 3.2: Data word exapansion

contains a much larger RAM, the data-page pointer here has been expanded to 8-bits from TMS32010's single or double bit versions. [20].

Since multiplication is the only instruction which supports immediate addressing, the DSP does not really support immediate addressing when it comes to any other operations, including ALU operations. Therefore, immediate addressing is not claimed to be supported by the DSP.

3.2.1 Direct addressing

Though direct addressing seems fairly straight forward in implementation, it involves a two-step process. In the first step, the data-page pointer register needs to be loaded with the value of the most significant or MSB 8-bits of the address, using a separate load data-page pointer instruction. The second step is to specify the remaining 7-bits of the address in the least significant or LSB 7-bits of the instruction, while resetting the eighth bit of the instruction to indicate direct addressing mode.

Hence, the MSB 8-bits of the RAM address is considered the page. Or in other words, the RAM has 256 pages, each page consisting of 128 words. This means that, once we load the page address, accessing any data within the page could be done in a single instruction. The main drawback of this system though is that every time we need to access a different page, we must use an additional instruction to load the page address. The advantage though it that the instruction word remains small, and hence the power efficiency is high compared to a bigger instruction word. Fig. 3.3 illustrates the working of direct addressing.

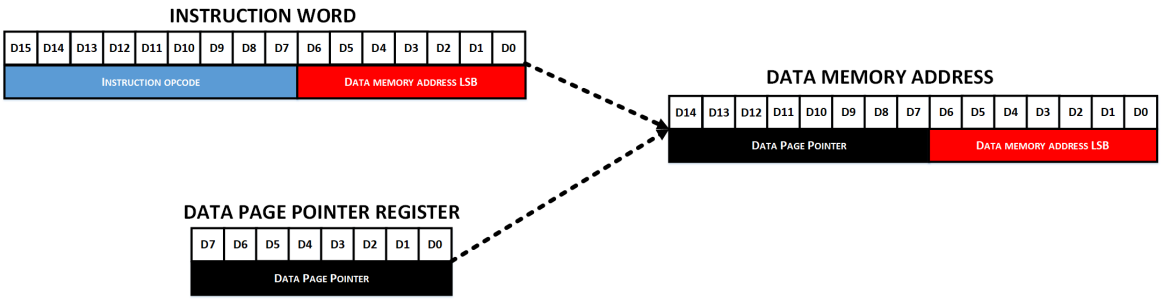

Figure 3.3: Direct addressing illustration

3.2.2 Indirect addressing

Indirect addressing is a multi-step process, since it accomplishes two things: selecting the next auxiliary register (AR) and incrementing/decrementing the current AR. The ARs hold address locations for the RAM data to be fetched, hence it is required to load them prior to using indirect addressing instructions.

The first step is to load one or more ARs with the value of the desired address location/s. Next, the AR pointer (ARP) needs to be set to the AR containing the next immediate address to be accessed. Later, the value of this AR can be incremented/ decremented, to be ready the next time the same AR is accessed.

Comparing indirect and direct addressing, indirect addressing is an attractive choice if the same data or its immediate neighbor is accessed multiple times. Since indirect addressing happens via the ARs, similar to direct addressing the ARs initially need to be loaded with address locations that are to be accessed. The advantage here is that these registers can be incremented/decremented every cycle, while also having the option of selecting which AR is to be accessed for the next operation.

Expanding the number of these registers is hence very helpful in numerous applications where consecutive data in multiple locations is required for the algorithm. The best example for such applications are image filters, where pixels within a 3x3 window of area are

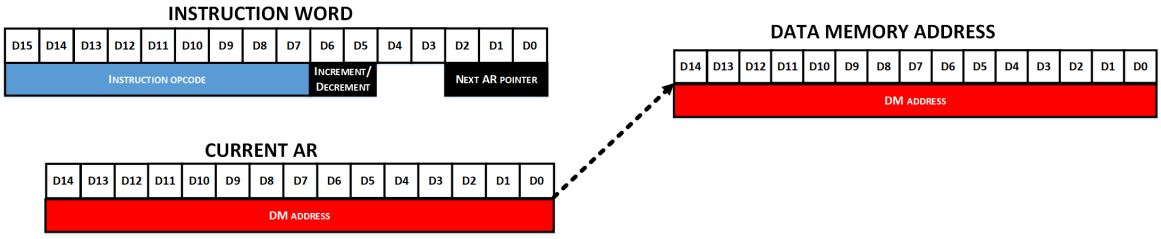

Figure 3.4: Indirect addressing illustration

usually used in numerous algorithm. Fig. 3.4 illustrates the working of indirect addressing.

3.3 Instruction opcodes and operation

The instruction set consists of a total of 50 instructions, including load/store, branching, data manipulation and SIMD instructions. Some instructions from the TMS32010 that are not implemented here are the table read and write, LTD, IN and OUT instructions. The Sections 3.3.1 and 3.3.2 list all the instructions and attempt to briefly describe their operations respectively.

3.3.1 List of instructions and corresponding opcodes

Table 3.1 shows the opcodes for all instructions. It is to be noted that all branching instructions will be followed by the jump address in the next instruction word, since it is not explicitly indicated in the table.

Table 3.1: List of Instructions and their opcodes

	Instr.	# Cyc	# IW	D 15	D 14	D 13	D 12	D 11	D 10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
1	MPY	1	1	0	0	0	0	0	0	0	0	M	D	D	D	D	D	D	D

Table 3.1: List of Instructions and their opcodes

	Instr.	**# Cyc**	**# IW**	**D 15**	**D 14**	**D 13**	**D 12**	**D 11**	**D 10**	**D9**	**D8**	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
		1										M	*	*	0	0	AR2	AR1	AR0
2	MPYK	1	1	1	0	1	0	0	0	0	0	K	K	K	K	K	K	K	K
3	MAC	1 1	1	0	0	0	0	0	0	1	0	M M	D *	D *	D 0	D 0	D AR2	D AR1	D AR0
4	OR	1 1	1	0	0	0	0	0	0	1	1	M M	D *	D *	D 0	D 0	D AR2	D AR1	D AR0
5	XOR	1 1	1	0	0	0	0	0	1	0	0	M M	D *	D *	D 0	D 0	D AR2	D AR1	D AR0
6	SPAC	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
7	SUB	1 1	1	0	0	1	S	S	S	S	S	M M	D *	D *	D 0	D 0	D AR2	D AR1	D AR0
8	SUBS	1 1	1	0	0	0	0	0	1	0	1	M M	D *	D *	D 0	D 0	D AR2	D AR1	D AR0
9	ADD	1 1	1	0	1	0	S	S	S	S	S	M M	D *	D *	D 0	D 0	D AR2	D AR1	D AR0
10	ADDS	1 1	1	0	0	0	0	0	1	1	0	M M	D *	D *	D 0	D 0	D AR2	D AR1	D AR0
11	AND	1 1	1	0	0	0	0	0	1	1	1	M M	D *	D *	D 0	D 0	D AR2	D AR1	D AR0
12	BU	2	2	0	0	0	0	0	0	0	1	0	0	0	0	1	1	1	1
13	BANZ	2	2	0	0	0	1	1	0	0	0	M	0	0	0	0	0	0	0

Table 3.1: List of Instructions and their opcodes

	Instr.	**# Cyc**	**# IW**	**D 15**	**D 14**	**D 13**	**D 12**	**D 11**	**D 10**	**D9**	**D8**	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
		2										M	*	*	0	0	AR2	AR1	AR0
14	BGEZ	2	2	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1
15	BGZ	2	2	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0
16	BLEZ	2	2	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1
17	BLZ	2	2	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0
18	BNZ	2	2	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1
19	BV	2	2	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	0
20	BZ	2	2	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	1
21	LAC	1	1	0	1	1	S	S	S	S	S	M M	D *	D *	D 0	D 0	D AR2	D AR1	D AR0
22	LACK	1	1	1	0	1	0	0	0	0	1	K	K	K	K	K	K	K	K
23	LAR	1 1	1	1	1	0	0	0	AR	AR	AR	M M	D *	D *	D 0	D 0	D AR2	D AR1	D AR0
24	LARK	1	1	1	1	0	0	1	AR	AR	AR	K	K	K	K	K	K	K	K
25	LARP	1	1	1	0	1	0	0	0	1	1	0	0	0	0	0	K	K	K
26	LDP	1 1	1	0	0	0	0	1	0	0	1	M M	D *	D *	D 0	D 0	D AR2	D AR1	D AR0
27	LDPK	1	1	1	0	1	0	0	1	0	0	K	K	K	K	K	K	K	K
28	LT	1 1	1	0	0	0	0	1	0	1	0	M M	D *	D *	D 0	D 0	D AR2	D AR1	D AR0
29	LTA	1	1	0	0	0	0	1	0	1	1	M	D	D	D	D	D	D	D

Table 3.1: List of Instructions and their opcodes

	Instr.	**# Cyc**	**# IW**	**D 15**	**D 14**	**D 13**	**D 12**	**D 11**	**D 10**	**D9**	**D8**	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
		1										M	*	*	0	0	AR2	AR1	AR0
30	LTP	1 1	1	0	0	0	0	1	1	0	1	M M	D *	D *	D 0	D 0	D AR2	D AR1	D AR0
31	LTS	1 1	1	0	0	0	0	1	1	1	0	M M	D *	D *	D 0	D 0	D AR2	D AR1	D AR0
32	MAR	1 1	1	0	0	0	0	1	1	1	1	M M	D *	D *	D 0	D 0	D AR2	D AR1	D AR0
33	PAC	1	1	0	0	0	0	0	0	0	1	0	0	0	1	1	1	1	1
34	ROVM	1	1	0	0	0	0	0	0	0	1	0	0	1	0	1	1	1	1
35	SAC	1 1	1	1	0	0	S	S	S	S	S	M M	D *	D *	D 0	D 0	D AR2	D AR1	D AR0
36	SAR	1 1	1	1	1	0	1	0	AR	AR	AR	M M	D *	D *	D 0	D 0	D AR2	D AR1	D AR0
37	SOVM	1	1	0	0	0	0	0	0	0	1	0	0	1	1	1	1	1	1
38	NOP	1	1	0	0	0	0	0	0	0	1	0	1	0	0	1	1	1	1
39	ZAC	1	1	0	0	0	0	0	0	0	1	0	1	0	1	1	1	1	1
40	ZALH	1 1	1	0	0	0	1	0	0	1	1	M M	D *	D *	D 0	D 0	D AR2	D AR1	D AR0
41	ZALS	1 1	1	0	0	0	1	0	1	0	0	M M	D *	D *	D 0	D 0	D AR2	D AR1	D AR0
42	APAC	1	1	0	0	0	0	0	0	0	1	0	1	1	0	1	1	1	1

Table 3.1: List of Instructions and their opcodes

	Instr.	**# Cyc**	**# IW**	**D 15**	**D 14**	**D 13**	**D 12**	**D 11**	**D 10**	**D9**	**D8**	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
43	CMPS--IMD	1 1	1	0	0	0	1	0	1	0	1	M M	- *	- *	G/L G/L	- 0	- AR2	- AR1	- AR0
44	SUBS--IMD	1 1	1	0	0	0	1	0	1	1	0	M M	D *	D *	D 0	D 0	D AR2	D AR1	D AR0
45	ADDS--IMD	1 1	1	0	0	0	1	0	1	1	1	M M	D *	D *	D 0	D 0	D AR2	D AR1	D AR0
46	POP	1	1	0	0	0	0	0	0	0	1	0	1	1	1	1	1	1	1
47	PUSH	1	1	0	0	0	0	0	0	0	1	1	0	0	0	1	1	1	1
48	RET	1	2	0	0	0	0	0	0	0	1	1	0	0	1	1	1	1	1
49	CALL	2	2	0	0	0	0	0	0	0	1	1	0	1	0	1	1	1	1

3.3.2 Description of the operation of each instruction

The operation of all instructions is listed in Table 3.2. It can be observed from the table that each instruction has 4-stages of implementation, these are the four pipeline stages, which are explained in detail in Chapter 4.

Table 3.2: List of instructions and their operations

Sl no.	***Instruction***	***Formula***	***Example***
1	MPY	Treg * [dma]-> Preg	MPY dma MPY {*|*+|*-}, next ARP
2	MPYK	Treg * constant -> Preg	MPYK constant
3	MAC	Treg * [dma] -> Preg Acc + Preg -> Acc	MAC dma MAC {*|*+|*-}, next ARP
4	OR	(Acc | [dma]) & 0xffffffff -> Acc	OR dma OR {*|*+|*-}, next ARP
5	XOR	(Acc ^ [dma]) & 0xffffffff -> Acc	XOR dma XOR dma
6	SPAC	Acc - Preg -> Acc	SPAC
7	SUB	Acc - [dma]*2^{shift} -> Acc	SUB dma, shift SUB {*|*+|*-}, shift, next ARP
8	SUBS	Acc - [dma] -> Acc	SUBS dma SUBS {*|*+|*-}, next ARP
9	ADD	Acc + [dma]* 2^{shift} -> Acc	ADD dma, shift ADD {*|*+|*-}, shift, next ARP
10	ADDS	Acc + [dma] -> Acc	ADDS dma ADDS {*|*+|*-}, next ARP
11	AND	(Acc & [dma]) & 0x0000ffff -> Acc	AND dma AND {*|*+|*-}, next ARP
12	BU	[pma] -> PC	BU pma
13	BANZ	[Is AR(ARP) != 0]; Yes => [pma -> PC]	BANZ pma

Table 3.2: List of instructions and their operations

Sl no.	***Instruction***	***Formula***	***Example***
		No => [PC + 2 -> PC]	BANZ pma, {*|*+|*-}, next ARP
14	BGEZ	[Is (ACC) >= 0]; Yes => [pma -> PC] No => [PC + 2 -> PC]	BGEZ pma
15	BGZ	[Is (ACC) > 0]; Yes => [pma -> PC] No => [PC + 2 -> PC]	BGZ pma
16	BLEZ	[Is (ACC) <= 0]; Yes => [pma -> PC] No => [PC + 2 -> PC]	BLEZ pma
17	BLZ	[Is (ACC) < 0]; Yes => [pma -> PC] No => [PC + 2 -> PC]	BLZ pma
18	BNZ	[Is (ACC) != 0]; Yes => [pma -> PC] No => [PC + 2 -> PC]	BNZ pma
19	BV	[Is OV == 1]; Yes => [[pma -> PC] && [OV -> 0] No => [PC + 2 -> PC]	BV pma
20	BZ	[Is ACC == 0]; Yes => [pma -> PC] No => [PC + 2 -> PC]	BZ pma
21	LAC	[dma]*2^{shift} -> Acc	LAC dma, shift LAC {*|*+|*-}, shift, next ARP
22	LACK	constant -> Acc	LACK 8-bit positive constant
23	LAR	[dma] -> AR	LAR AR, dma LAR AR, {*|*+|*-}, next ARP
24	LARK	constant -> AR	LARK AR, 8-bit positive constant
25	LARP	constant -> ARP	LARP 3-bit constant

Table 3.2: List of instructions and their operations

Sl no.	***Instruction***	***Formula***	***Example***
26	LDP	[dma] & 0xff -> data page pointer	LDP dma LDP {*|*+|*-}, next ARP
27	LDPK	constant -> data page pointer	LDPK 8-bit constant
28	LT	[dma] -> Treg	LT dma LT {*|*+|*-}, next ARP
29	LTA	[dma] -> Treg Acc + Preg -> Acc	LTA dma LTA {*|*+|*-}, next ARP
30	LTP	[dma] -> Treg Preg -> Acc	LTP dma LTP {*|*+|*-}, next ARP
31	LTS	[dma] -> Treg Acc - Preg -> Acc	LTS dma LTS {*|*+|*-}, next ARP
32	MAR	Modifies AR(ARP), and ARP as specified	MAR dma MAR {*|*+|*-}, next ARP
33	PAC	Preg -> Acc	PAC
34	ROVM	0 -> OVM status bit	ROVM
35	SAC	(Acc) $*2^{\text{shift}}$ -> [dma]	SAC dma, shift SAC {*|*+|*-}, shift, next ARP
36	SAR	AR -> [dma]	SAR AR, dma SAR AR, {*|*+|*-}, next ARP
37	SOVM	1 -> overflow mode (OVM status bit)	SOVM
38	NOP	N/A	N/A
39	ZAC	0 -> Acc	ZAC

Table 3.2: List of instructions and their operations

Sl no.	***Instruction***	***Formula***	***Example***
40	ZALH	0 -> Acc[15:0] [dma] -> Acc[31:16]	ZALH dma ZALH {*|*+|*-}, next ARP
41	ZALS	0 -> Acc[31:16] [dma] -> Acc[15:0]	ZALS dma ZALS {*|*+|*-}, next ARP
42	APAC	Acc + Preg -> Acc	APAC
43	CMPSIMD	Acc[7:0] v/s dma[7:0] -> Acc[7:0] Acc[15:8] v/s dma[15:8] -> Acc[15:8] Acc[23:16] v/s dma[23:16] -> Acc[23:16] Acc[31:24] v/s dma[31:24] -> Acc[31:24]	CMPSIMD dma CMPSIMD {*|*+|*-}, next ARP
44	SUBSSIMD	Acc[7:0] - (dma[7:0]) -> Acc[7:0] Acc[15:8] - (dma[15:8]) -> Acc[15:8] Acc[24:16] - (dma[24:16]) -> Acc[24:16] Acc[31:24] - (dma[31:24]) -> Acc[31:24]	SUBSIMD dma SUBSIMD {*|*+|*-}, next ARP
45	ADDSSIMD	Acc[7:0] + (dma[7:0]) -> Acc[7:0] Acc[15:8] + (dma[15:8]) -> Acc[15:8] Acc[24:16] + (dma[24:16]) -> Acc[24:16] Acc[31:24] + (dma[31:24]) -> Acc[31:24]	ADDSIMD dma ADDSIMD {*|*+|*-}, next ARP
46	PUSH	Acc -> Stack	PUSH
47	POP	Stack -> Acc	POP
48	CALL	PC -> Stack [pma] -> PC	CALL L2
49	RET	PC -> Restore from Stack	RET

Chapter 4

DSP Pipeline and Read/Write RAM buffer wrapper implementation

Speed of computation has been the biggest challenge for any digital processor since its invention, measured as the number of instructions that can be executed per second. In the processor world, it has been established from years of experimentation and observation there are only two factors which can significantly increase the speed computing. The first factor being evolution of fabrication technology and the second being better computer architecture, which has famously been marketed by Intel's tick-tock processor model [21]. Fabrication technology understandably is of enormous complexity, since some of the topics involved are chemical reactions, photonics, material science and device physics.

CPU architecture planning makes a significant impact in its speed enhancement. The key factor in achieving computation speed is concurrency: performing as many operations as possible, simultaneously. Concurrency though has two very important implementations, namely pipelining, and parallelism. Although rooted in the same origins, often hard to distinguish in practice, the two terms are discernibly different in their general approach

[22].

Looking at a typical DSP MAC instruction, operands need to be fetched from the memory, multiplied, while the previous product is added to the accumulator and address register is post incremented/decremented. It is obvious that, to accomplish all these sequential functions it would take multiple clock cycles if the DSP is not pipelined [23].

While pipelining effectively speeds up the computation, programmability takes a serious hit, if not done properly. This may also result in loosing some instruction cycles due to data dependency hazards. When a programmer writes an assembly program, it is assumed that every instruction completes before the next instruction begins. This must be ensured by carefully designing the pipeline, such that the DSP should appear as if it were not pipelined even though it is [23].

4.1 Pipeline implementation

In a good pipeline design, extensive pipelining with parallel architecture capability has to be implemented,while ensuring programmabilty is not impacted due to dependency hazards. This requires a system-level understanding of the DSP along with careful planning of each pipeline stage [24].

In chapter 2, the DSP architecture had a brief look at parallelism with SIMD implementation in the ALU. Here, a clear description of pipelining is presented. This chapter explains how exactly each task is split into pieces, while being tackled simultaneously.

4.1.1 Pipeline stages

Before planning the pipeline, it is very important to understand the sequence of events happening in the DSP and data arrival time. Some very important points to remember

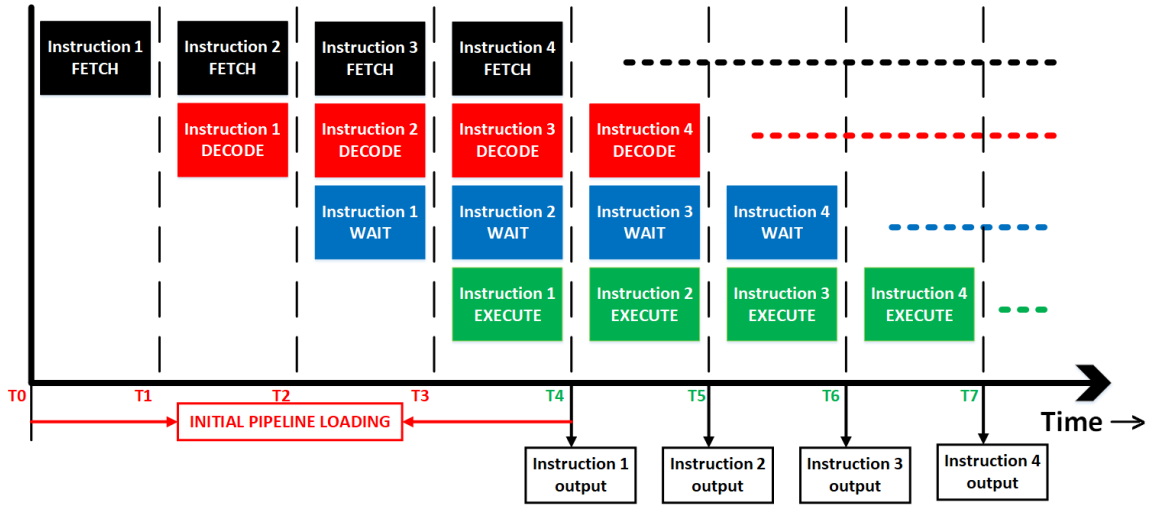

Figure 4.1: Pipeline stages and implementation

are:

1. It takes at least one clock cycle to fetch data from the ROM.
2. The DSP requires one clock cycle to decode the instruction coming from the ROM.
3. Data transfer to and from the RAM also takes at least one clock cycle.
4. Most instructions in the DSP use direct memory access (DMA), hence most of them will have to read data from the RAM.
5. Most instructions are to be executed within a single clock cycle.

Considering all the points mentioned above, the DSP pipeline has been divided into 4 stages. The name and function of each stage is described in Fig. 4.1.

1. The first stage is *FETCH*, where operand is fetched from the ROM. The program counter is updated by this stage, starting as soon as the DSP is switched on and reset.

2. The second stage is *DECODE*, where the fetched instruction is decoded. This stage is responsible for generating the RAM address and corresponding handshake signals, if necessary.

3. The third stage is *WAIT*, where the DSP waits for data to be fetched from the RAM, and feeds the read data into the execution unit. Also, updating AR and ARP is done at this stage.

4. The fourth stage is *EXECUTE*, where all arithmetic operations are performed by the DSP.

Fig.4.1.1 illustrates how the pipeline works in the DSP for the first 4 instructions. Assuming that the DSP is reset at T0, it is observed that the DSP does not execute the first instruction until T4. However, after T4, for every cycle there will be an output. Hence, technically all single cycle instructions have a latency of 4 cycles, though they just take a single cycle to execute.

4.1.2 Pipeline design for non-branching instructions

The pipelining of non-branching instruction is straight forward for all read instructions. Write instructions however need to be modified slightly because of the way the RAM memory works and DMA design of the DSP.

Fig.4.2 illustrates the pipeline operation for DMA read instructions, where all four instructions are assumed to be DMA read instructions. The steps followed at each pipeline stage of the implementation of DMA read instruction are listed below:

1. Fetch stage: The fetch stage here reads the instruction from the ROM and stores it in an internal register for the next stage. It also increments the value of PC by one, so that the next instruction is fetched in the following cycle.

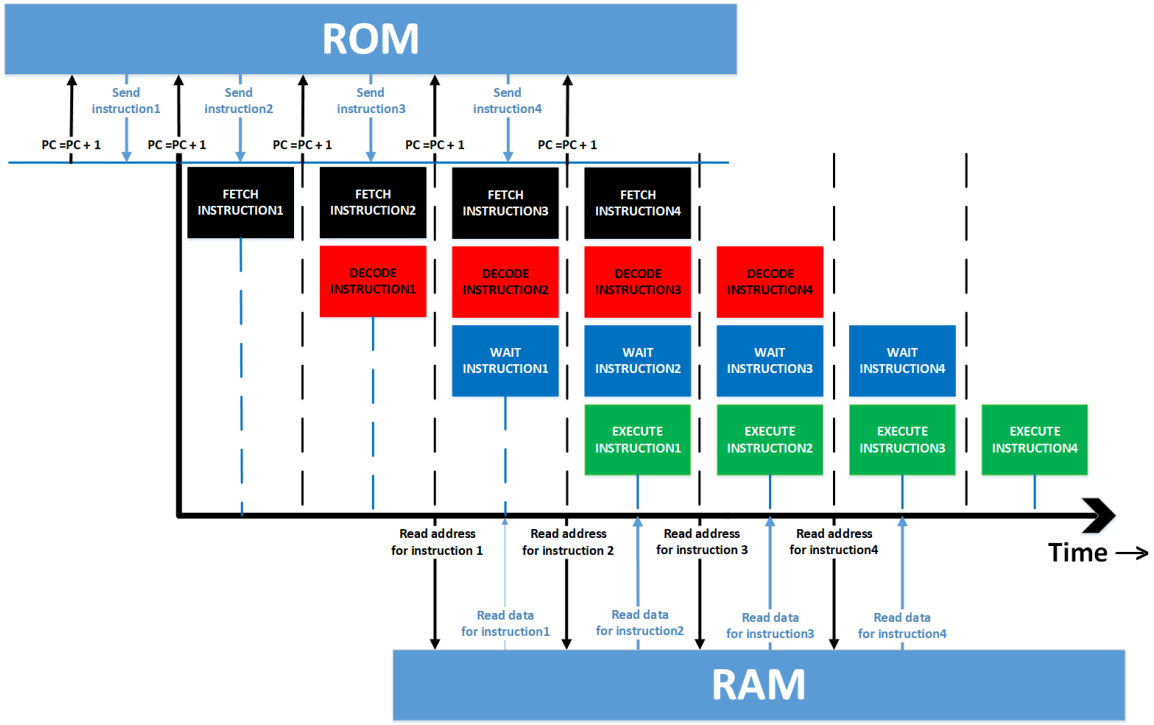

Figure 4.2: Pipeline example for memory read instructions

2. Decode stage: The decode stage here decodes whether the instruction is direct addressing or indirect addressing. It is also responsible to setup the appropriate handshake signals for memory read and generate read address after decoding the instruction.

3. Wait stage: In the case of direct addressing, the wait stage does nothing. However, this stage takes care of updating AR and ARP registers for indirect addressing.

4. Execute stage: By this stage, the memory read operation would have finished. Hence, the fetched data is now used by the execution unit to compute. By the end of this cycle, the output is either stored in the Product register or the accumulator, depending on the instruction executed.

Fig. 4.3 illustrates the pipeline operation for 4 instructions, where the second and third

are DMA write instructions, while the first and fourth are DMA read instructions. The steps followed at each pipeline stage of the implementation of DMA write instruction are listed below:

1. Fetch stage: This stage is the exact same as DMA read fetch stage. Hence, the instruction is read from the ROM and passed on to the decode stage, while incrementing the value of PC by one for fetching the next instruction.

2. Decode stage: The instruction is decoded, setting up the write address according to direct/indirect addressing. Appropriate handshake signals for memory write are generated.

3. Wait stage: By the time this stage is completed, the appropriate write data needs to be ready. To accomplish this, appropriate changes in the architecture have been made at this stage to shift the output of the results of the fourth cycle of the previous instruction in the case of SAC or store accumulator instruction, in case the previous instruction is an ALU operation. The updating of AR and ARP for indirect addressing also it the responsibility of this stage.

4. Execute stage: During this stage, the memory write operation is performed by the read/write RAM buffer wrapper.

4.1.3 Pipeline design for unconditonal branching instructions

As observed from Table 3.1, branching instructions are two-cycle instructions, and all except return or RET require two-instruction words. From Chapter 3, it is also established that the first instruction word contains the instruction op-code, while the second contains the jump or branch address.

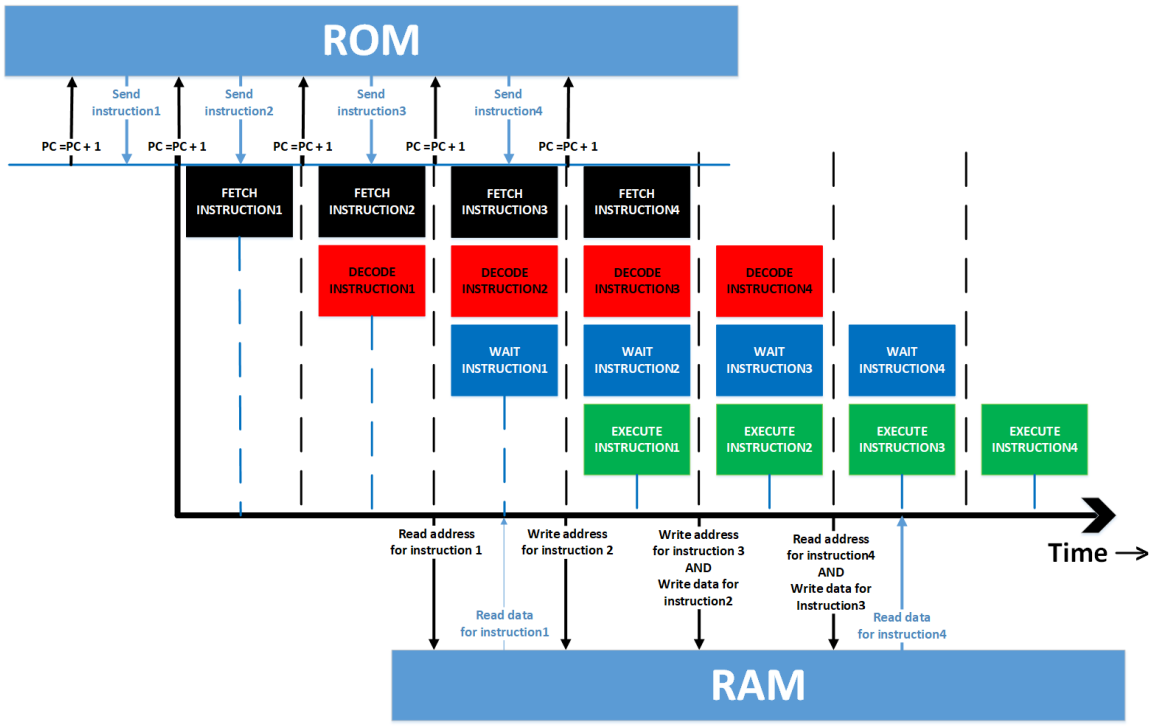

Figure 4.3: Pipeline example for memory write instructions

During the execution of two-word branch instruction, all stages of the pipeline are stalled while reading the second word, since it is not an instruction. Appropriate changes are made at every pipeline stage to make sure that the second word is not read as an instruction, but stored as jump address.

There are two types of branching instructions, namely conditional and unconditional branching instructions. Unconditional branching instructions are BU (branch unconditional), CALL (call) and RET (return), where branch must always be taken. Conditional branching instructions are where the branching decision is made based on the valuation of a condition.

Figure 4.4 shows a pipeline implementation example of unconditional branch instruction. The steps followed at each pipeline stage of the implementation of unconditional branch instructions are listed below:

1. Fetch stage: The DSP during this stage reads the instruction from the ROM and increments PC by one, just like all other instructions. However, the fetch is stalled in order to read the jump or branch address.

2. Decode stage: The instruction is decoded, and call registers are accordingly modified for call and return instructions, while the jump address is read from the program memory and fed to the PC.

3. Wait stage: As the unconditional instruction has already been executed at this point, this stage does almost nothing. However, for call and return instructions, the stack pointer is stored and restored respectively at this stage.

4. Execute stage: This stage does performs no task since the instruction has already accomplished its purpose.

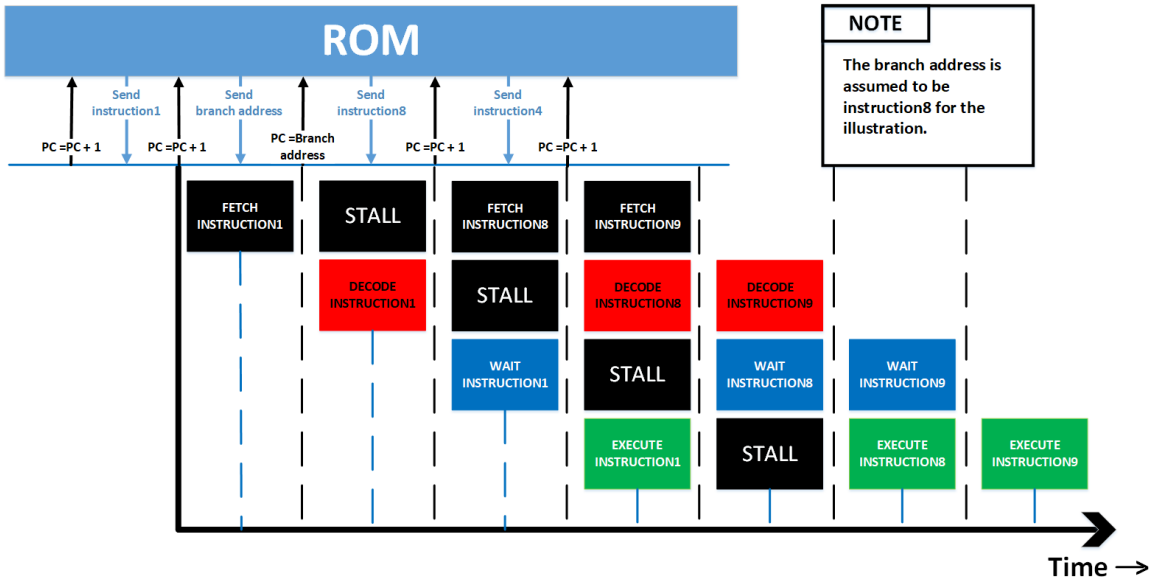

Figure 4.4: Pipeline example for unconditional branching

4.1.4 Pipeline design for conditional branching instructions

Before looking at the implementation of conditional branching, it is necessary to understand how the instruction works in practice. Since almost all conditional branching instructions rely either on status flags resulting from an ALU operation, or the ALU result itself, timing and data dependency wise, the worst-case scenario of the previous instruction being an ALU operation is assumed before approaching to design the pipeline stages.

With the assumption that the previous instruction is an ALU operation, the outcome of the branching condition is not known until the operation is complete. From the pipeline design for ALU operations, or DMA read operations, discussed in Section 4.1.2, it is clear that execution happens only at the last pipeline stage. Hence, the branching decision cannot be made until after the fourth cycle of the previous instruction has been executed. However, after stalling the pipeline to read the branch address, before the third stage of the pipeline of the conditional branch instruction, the DSP should already know where to

fetch the next instruction from. In other words, during the second pipeline stage, the PC needs to be updated to the next program address to fetch from. This results in a dilemma, as the branching decision needs to be made at the second stage of the pipeline, however the decision is not available until the fourth stage.

There are two solutions to this problem:

Solution 1: Decide to not take the branch, and make necessary changes if the condition turns out to be true.

Solution 2: Predict the branch, and pay the penalty of two cycles if wrong by making necessary changes in the case of a wrong prediction.

Though solution 2 is a better option and has numerous methods of execution, even the simplest branch predictor requires a lot of additional hardware and planning. To keep the DSP design simple, solution 1 is considering in this design.

Figures 4.5 and 4.6 illustrate both cases of the working of pipeline for conditional branch instructions, the first case where the condition evaluates to be false, and the second case where the condition evaluates to be true. In both cases, the first instruction is assumed to be the evaluation instruction, hence the branch/jump condition is evaluated based on its outcome. The second instruction is the conditional branch instruction, while the last two instructions are unconditional instructions.The pipeline plan of action for each stage is listed below:

1. Fetch stage: The DSP during this stage reads the instruction from the ROM and increments PC by one, just like all other instructions. Fetching of the next instruction is stalled to read the jump or branch address.

2. Decode stage: The instruction is decoded, and the jump address is read from the program memory and stored until execution stage. However, the value of PC is

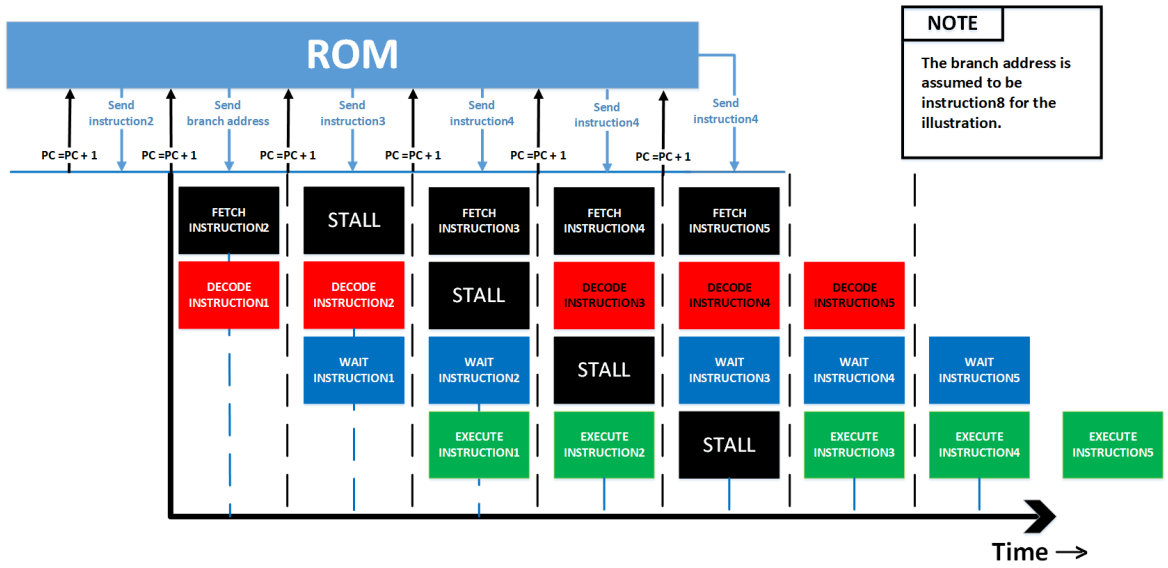

Figure 4.5: Pipeline implementation example for conditional branch instruction, when condition is false

incremented by one for the pipeline to work smoothly, and not waste any cycles in case the branch evaluates to be false.

3. Wait stage: This stage does performs no task.

4. Execute stage: The branching condition is evaluated at this stage. In case the condition evaluates false, no change is made to the pipeline flow. However, if the condition evaluates to be true, the value of PC is updated to the jump address, resulting in the wastage of the computations in the previous two cycles. It is very important to undo any modifications done in the previous two cycles and also stall the pipeline accordingly, to make sure no unwanted data is propagated, in case the jump is taken.

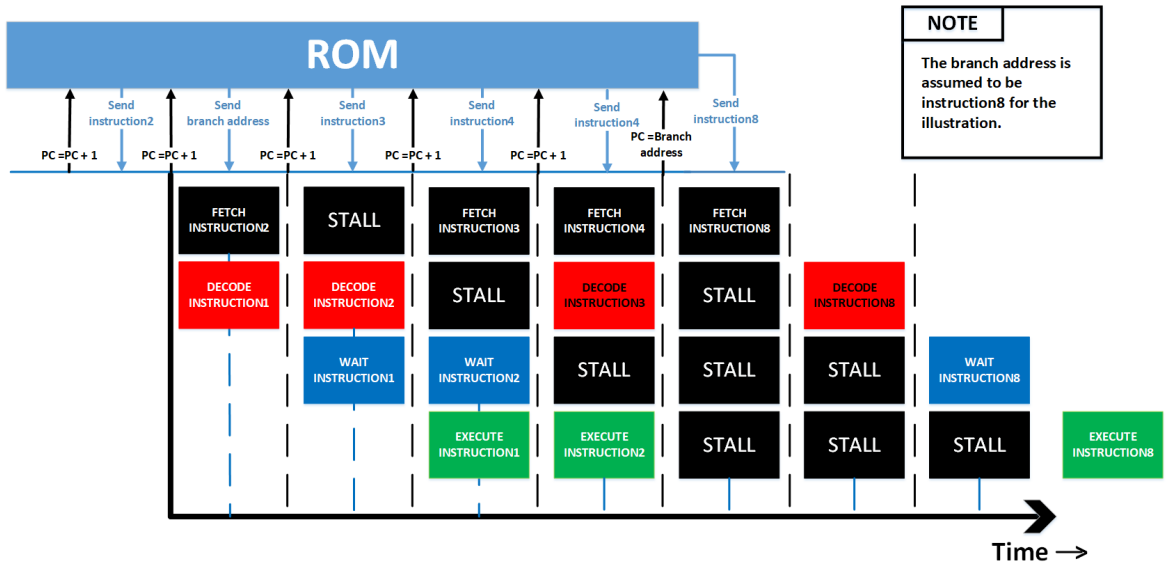

Figure 4.6: Pipeline implementation example for conditional branch instruction, when condition is true

4.2 Read/write RAM buffer wrapper

DSPs have much higher memory bandwidth and use lot more memory-to-memory instructions, when compared to traditional processors [25]. While most DSPs tackle this problem using small, fast and simple parallel memory banks, it is very difficult to design compilers and the power consumption increases significantly for such DSPs [18]. Since it has been established that data memory access is very important in DSPs, it is crucial to ensure that memory access is quick and effective, while keeping the power consumption low. Hence, both, data and address memories have been clocked at the same speed as the DSP, in an effort to keep the total power consumption low.

Taking a brief look at the pipeline implementation described in the Section 4.1.2, it can be observed that there exists a huge problem in the case of RAM memory writes, since the write data is provided a cycle after write address generation. Since the pipelining and data memory addressing of the DSP design implemented in this paper is very different from

TMS32010, even though the ISA is almost the same, this problem is not observed in the case of TMS32010. This is mainly because TMS32010 had its memory clocked to at least twice the speed of the DSP itself. This is evident from some of the instructions in its ISA, which have obviously not been implemented in this DSP. A good example for this is the LTA instruction, which featured multiple memory transactions within a single clock cycle. [20]

Section 4.2.1 describes what problems were faced due to clocking the memory at the same speed as the processor, and Section 4.2.2 describes how the problem has been resolved using the read/write RAM buffer wrapper.

4.2.1 RAM read/write problem description

Looking at the pipeline implementation in the case of data memory or DMA write operations in Section 4.1.2, it is observed that write address and handshaking signal generation happens at stage 2 or decode stage, while the write data is sent to the data memory in the next stage, which is stage 3 or the wait stage. However, the RAM requires the address, handshaking signals, along with the data to be written, all within the same cycle. This is not possible with the pipeline design implemented in this paper, since the write data may be computed in last stage of the pipeline of the previous instruction.

Hence, since is in not possible to make sure that the RAM receives the write data at the correct cycle, a buffer layer has been designed to effectively facilitate the data-flow. The buffer layer is simple in design and implementation, using minimal hardware required to serve the purpose, since other easy solutions involve using a faster clock for the memory, leading to an increase in power consumption.

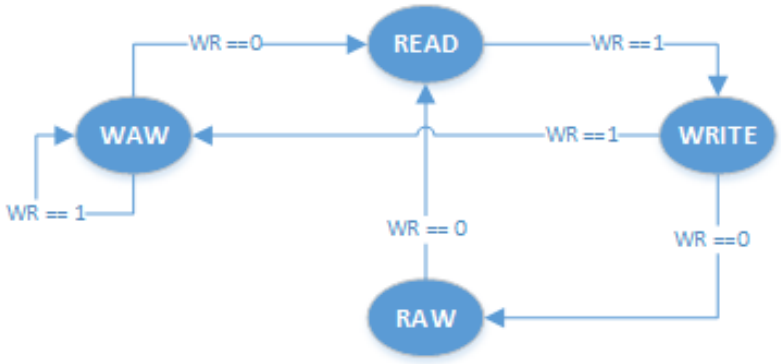

Figure 4.7: Read/write RAM buffer wrapper state machine

4.2.2 Design and implementation of read/write buffer wrapper

The wrapper is designed with a simple goal: delay the data memory write operation by a single cycle, while seamlessly providing the correct data whenever necessary. Translating this to a plan of action, the following procedures were followed:

1. For every write operation, store the address and corresponding data.

2. For every read, check the address. If it matches the buffer address, transfer the contents of the buffer data as the output to the DSP. Else, make the necessary arrangements to fetch the data directly from the RAM, and send it to the DSP.

3. For every other write operation following the first, store the buffer data onto RAM and update the buffer address and data with the corresponding new values.

Fig. 4.7 illustrates the state-machine for the read/write buffer wrapper. The state machine consists of a total of four states depending on the type of operation involved. Since write is implemented as a two-stage operation in the pipeline, the following instruction also needs to be accounted for within the read/write buffer wrapper. A brief explanation of the implementation is described below, detailing the operation of each state:

1. Read state: Read state is also the idle state. If the read address matches the buffer address register contents, the buffer data is transferred to the DSP. However, if the read address is different from the buffer address register contents, the required data is fetched from RAM and transferred to the DSP within the next clock cycle.

2. Write state: A single bit flag is used to keep track of whether the buffer data has been transferred onto the RAM or not. Every time a new data arrives, if data is present in the data buffer register, it is transferred to the RAM address corresponding to the buffer address, which is retrieved from the buffer address register. This is followed by storing the write address in the buffer address register.

3. RAW state: In the RAW state or read-after-write state, the write data is stored in the buffer data register. Also, all functions in the read state are performed here as well.

4. WAW state: In the WAW state or the write-after-write state, the write data is directly sent to the RAM, at the address location corresponding to the buffer address register. The new write address is now stored in the buffer address register.

Chapter 5

Median filter design

Image processing and filtering is an area where DSPs have been used extensively since their invention. In the recent years however, more complex image processing have been handled by GPUs or graphical processsing units mainly due to their hardware parallelism and enormous amount of data required to be processed. However, numerous image filtering applications are still use DSPs, but with multiprocessor type configuration.

Taking a brief look at image data, it is usually represented by the amount of Red, Green and Blue (RGB) colors over a fixed area of a preset number of very small points called pixels. The common representation of a standard dimension image is 24-bit RGB values per pixel, over an area of (720 x 576) pixels. Most simple DSPs are 16-bit fixed-point architectures, hence to handle image data they would require two data words per pixel. Expanding the data word to at least 24-bits hence could result in further applications in image handling and processing.

The following sections of this chapter present a simple application of the designed DSP, to showcase the merits of its enhancements over the TMS32010 by implementing a median filter. Section 5.1 presents an overview of the median filter by explaining how a median filter

works. Section 5.2 discusses the median filter algorithm design and its implementation.

5.1 Median filter overview

Median filters are non-linear digital filters used widely to get rid of salt and pepper noise. The implementation of the median is quite is simple and straight forward. Considering a 3x3 window of pixels of an image, the following steps are followed to find the median:

Step 1: Arrange the pixels one after the other.

Step 2: Rearrange the pixels in an ascending or descending order.

Step 3: Pick the central value of the arranged pixels, which is the fifth pixel in this case. This will be the median.

While the median filter implementation looks like a simple two-step process, it takes a significant amount of effort to arrange the pixels in ascending or descending order, as every pixel needs to be compared to every other pixel, and this needs to be done sequentially to keep track of the order of their arrangement.

Fig.5.1 illustrates the working of a median filter. In the figure, P1, P2, P3, P4, P5, P6, P7, P8 and P9 are pixel values of the 3x3 window from the image. After step 2, note that the new pixel values P1‘, P2‘, P3‘, P4‘, P5‘, P6‘, P7‘, P8‘ and P9‘ indicated in the figure represent the rearranged pixel values.

5.2 Median filter design and implementation

The median filter algorithm design for the 3×3 pixel window is explained in figure 5.2. The figure self-explanatory and hence clearly explains the algorithm which has been implemented in DSP assembly language.

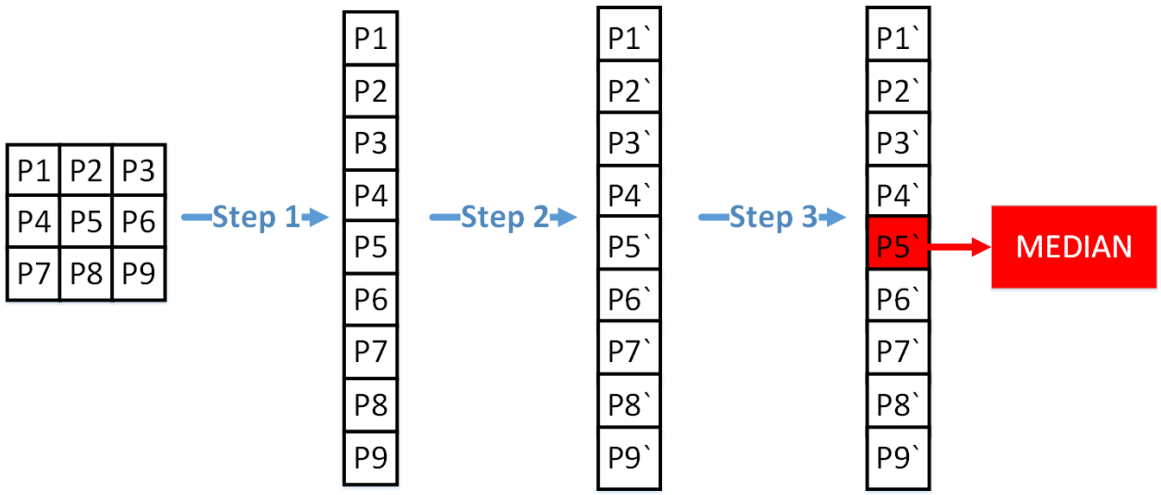

Figure 5.1: Median filter working illustration

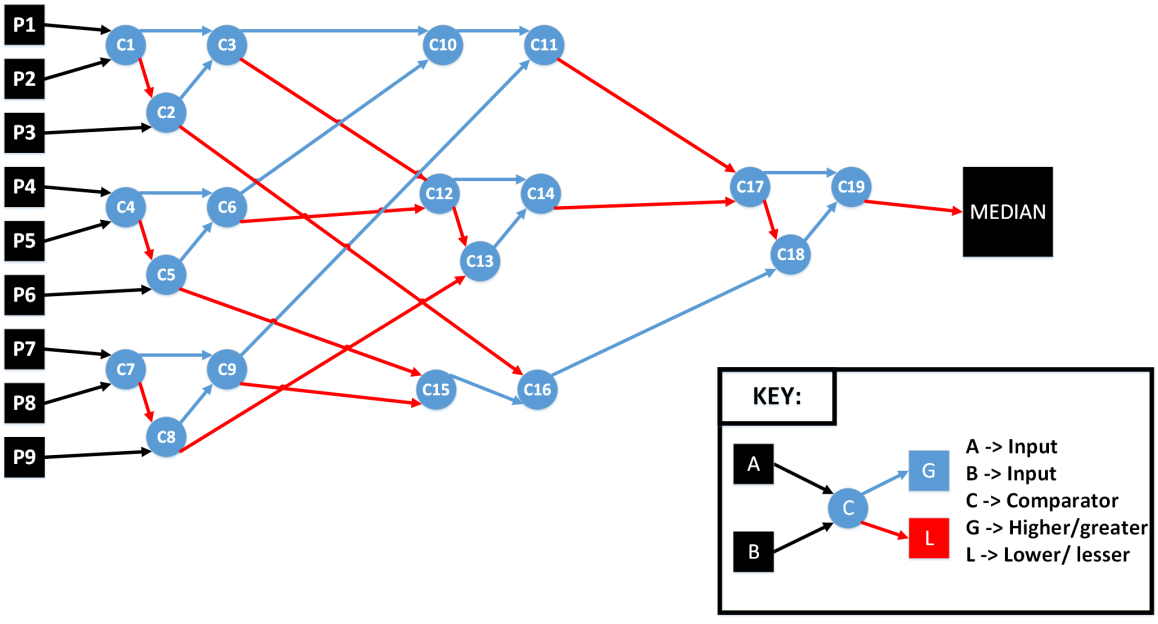

Figure 5.2: Median filter algorithm

The implementation of this algorithm on an image is done by moving the 3×3 window from the top-left corner of the image across all columns, and soon as the median for the first row has been computed, the window is moved to the next row. This is repeated until the last row is computed. Fig. 5.3 illustrates the implementation of algorithm on an image. The first part of the figure shows the 3×3 window placement while computing the first median, the second part shows the window placement while computing the second median and the third part shows the window placement while computing the median for the second row. This window placement pattern is repeated until all the medians are computed. It is worth noting that the output image will lose two rows and two columns, with this implementation.

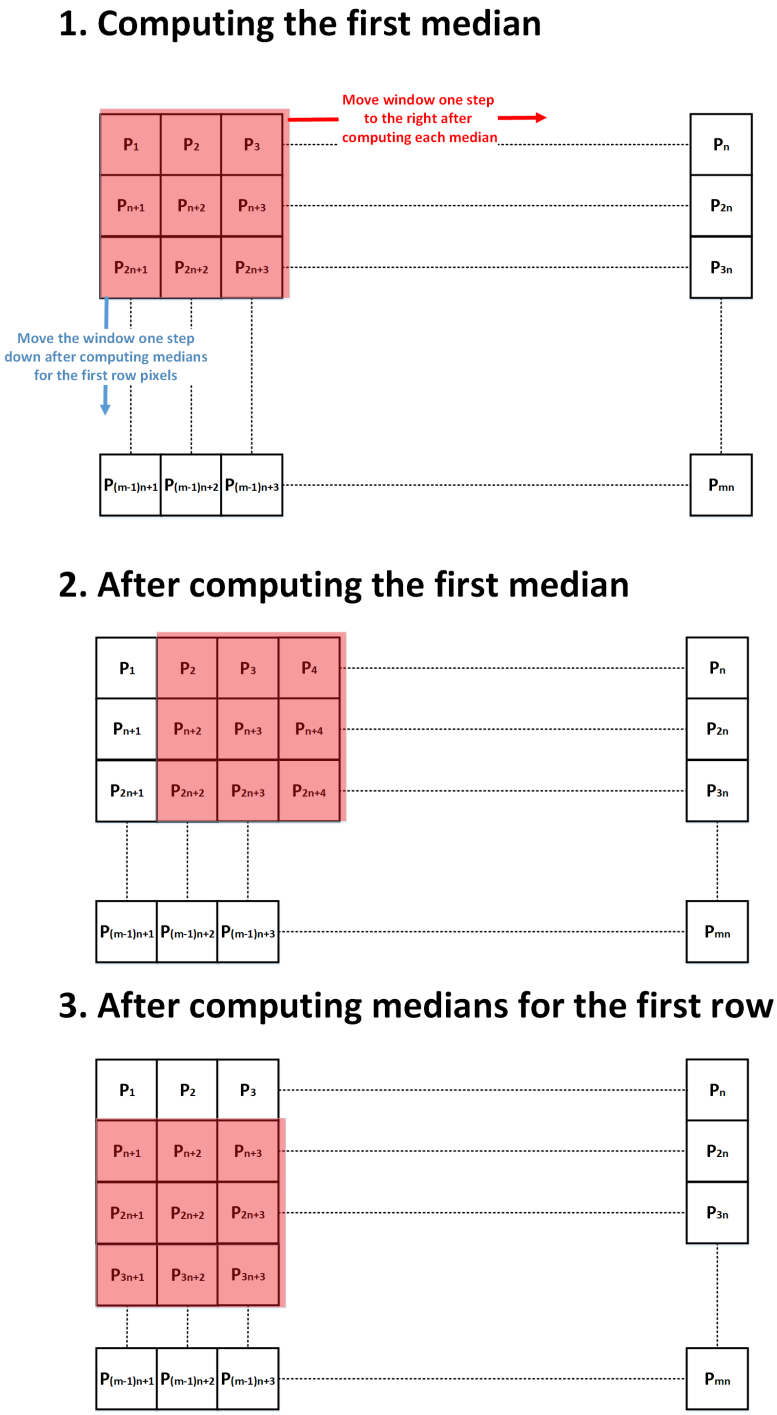

Figure 5.3: Median filter algorithm implementation illustration for a 3×3 window

Chapter 6

Results

This chapter discusses the results from this project, as well as future work that could be completed.

6.1 Results

The DSP design was synthesized using Synopsys Design Compiler at 180 nm technology nodes from TMSC. Cadence Virtuoso Suite was used for design, debuging and simulation of the design. Table 6.1 gives the synthesis results for the post-scan netlist of the design, when synthesized at 50MHz.

Due to time constriants, it was not possible to fully verify the DSP design. The DSP however has been verified to work at gate level, where most of its instructions and numerous branching dependencies have been tested. The basic median filter algorithm was designed in assembly language and verified to work on the DSP. Details including the Assembly code that was used for testing the DSP have been included in Appendix A.

Table 6.1: Synthesis results

Area (μm^2)	Noncombinational area	181298
	Combinational area	167021
	Buf/ Inv area	9027
	Total cell area	348320
Power (mW)	Internal Power	9.4111
	Switching Power	1.6481
	Leakage Power	1.4210
	Total	11.0607
Timing (ns)	Data arrival time	18.1474
	Slack	1.4799
DFT Coverage	Test coverage	99.92%

Chapter 7

Conclusions and future work

7.1 Conclusion

The DSP design has been successfully implemented, verified for the instructions mentioned in Appendix A and synthesized at 50MHz. The design was kept simple, since its ISA and architecture have been based on the TMS32010. Power efficiency was achieved by running the memory at the same speed as the DSP. A median filter algorithm was designed in assembly, simulated at gate-level and verified to work on the DSP within 100 instructions, demonstrating that the enhanced SIMD instructions could be used for median filter computation, hence proving that the DSP is capable of handling simple multimedia applications.

7.2 Future work

Since the DSP was designed in a very short span of time, testing the DSP thoroughly could not be completed. It is necessary to completely test the DSP before attempting to use it in an application, hence this would be the the first thing to work on. The median

filter algorithm described in chapter 5, though successfully implemented, could not be tested with a noisy image due to time constraints. Doing this would demonstrate the capabilities of the DSP, and a comparison of the results with a similar implementation on the TMS32010 would prove the claims presented in the paper.

Designing a compiler would certainly be necessary and the next step to work on. Another interesting enhancement would be the design of a parallel-processor system using mulitple DSPs for advanced imaging applications.

References

[1] B. Marr, "Big data: 20 mind-boggling facts everyone must read."

[2] T. Jamil, "Risc versus cisc," *Ieee Potentials*, vol. 14, no. 3, pp. 13–16, 1995.

[3] W. P. Hays, "Dsps: Back to the future," *Queue*, vol. 2, no. 1, p. 42, 2004.

[4] D. Zuras, M. Cowlishaw, A. Aiken, M. Applegate, D. Bailey, S. Bass, D. Bhandarkar, M. Bhat, D. Bindel, S. Boldo *et al.*, "Ieee standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.

[5] S. W. Smith, *The scientist and engineer's guide to digital signal processing*. California Technical Pub., 1999.

[6] C. Inacio and D. Ombres, "The dsp decision: Fixed point or floating?" *IEEE Spectrum*, vol. 33, no. 9, pp. 72–74, 1996.

[7] S. Smith, *Digital signal processing: a practical guide for engineers and scientists*, S. Smith, Ed. Newnes, 2013.

[8] G. Frantz, "Signal core: A short history of the digital signal processor," *IEEE Solid-State Circuits Magazine*, vol. 4, no. 2, pp. 16–20, 2012.

[9] E. J. Tan and W. B. Heinzelman, "Dsp architectures: past, present and futures," *ACM SIGARCH Computer Architecture News*, vol. 31, no. 3, pp. 6–19, 2003.

[10] A. Abnous and N. Bagherzadeh, "Pipelining and bypassing in a vliw processor," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 6, pp. 658–664, 1994.

[11] J. Glossner, J. Moreno, M. Moudgill, J. Derby, E. Hokenek, D. Meltzer, U. Shvadron, and M. Ware, "Trends in compilable dsp architecture," in *Signal Processing Systems, 2000. SiPS 2000. 2000 IEEE Workshop on.* IEEE, 2000, pp. 181–199.

[12] C. Choo, J. Chung, J. Fong, and S. E. Cheung, "Implementation of texas instruments tms32010 dsp processor on altera fpga," in *Global Signal Processing Expo & Conf. San Jose State University*, 2004.

[13] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach.* Elsevier, 2011.

[14] S. L. Harris and D. M. Harris, *Digital Design and Computer Architecture: ARM Edition.* Morgan Kaufmann, 2016.

[15] A. David and H. John, "Computer organization and design: the hardware/software interface," *San mateo, CA: M organ Kaufmann Publishers*, vol. 1, p. 998, 2005.

[16] K. Ngan, A. Kassim, and H. Singh, "Parallel image-processing system based on the tms32010 digital signal processor," *IEE Proceedings E (Computers and Digital Techniques)*, vol. 134, no. 2, pp. 119–124, 1987.

[17] D. Holburn and I. Sommerville, "A high-speed image processing system using the tms32010," *Software & Microsystems*, vol. 4, no. 5, pp. 102–108, 1985.

[18] E. A. Lee, "Programmable dsp architectures. i," *IEEE ASSP Magazine*, vol. 5, no. 4, pp. 4–19, 1988.

[19] G. Araujo, A. Sudarsanam, and S. Malik, "Instruction set design and optimizations for address computation in dsp architectures," in *Proceedings of the 9th international symposium on System synthesis*. IEEE Computer Society, 1996, p. 105.

[20] T. Instruments and P. Strzelecki, *TMS32010 User's Guide*. Texas Instruments, 1983.

[21] T. Jain and T. Agrawal, "The haswell microarchitecture-4th generation processor," *International Journal of Computer Science and Information Technologies*, vol. 4, no. 3, pp. 477–480, 2013.

[22] P. M. Kogge, *The architecture of pipelined computers*. CRC Press, 1981.

[23] E. A. Lee, "Programmable dsp architectures. ii," *IEEE ASSP Magazine*, vol. 6, no. 1, pp. 4–14, 1989.

[24] E. Lee and D. Messerschmitt, "Pipeline interleaved programmable dsp's: Architecture," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 35, no. 9, pp. 1320–1333, 1987.

[25] N. H. Weste and D. Harris, *CMOS VLSI design: a circuits and systems perspective*. Pearson Education India, 2015.

Appendix I

Source Code

I.1 RTL source code

I.1.1 DSP top level module

```
1  //
       ////////////////////////////////////////////////////////////////////////////////
2  // Author         : Shashank Simha
3  // Date           : 12/12/2017
4  // University     : Rochester Institute of Technology
5  // Description    : This is a part of the DSP implemented for grad project
6  //
       ////////////////////////////////////////////////////////////////////////////////
7  module DSP_Version1 (
8              reset,
9              clk,
10             scan_in0,
11             scan_en,
12             test_mode,
13             scan_out0,
14             SW_pin, Display_pin,
15             DM_out, CEN, wr_data, DM_Addr, DM_in, OEN,            //ram_buffer
16             PC, PM_out                                            //rom
17         );
18
19
```

```
20 input [4:0] SW_pin;                          // Four switches and one push-button
21 output [7:0] Display_pin;        // 8 LEDs
22
23 input
24     reset,                       // system reset
25     clk;                         // system clock
26
27 input
28     scan_in0,                    // test scan mode data input
29     scan_en,                     // test scan mode enable
30     test_mode;                   // test mode select
31
32 output
33     scan_out0;                   // test scan mode data output
34
35 ////////////RAM ports//////////////////
36 output [14:0] DM_Addr;
37 output [31:0] DM_in;
38 input [31:0] DM_out;
39 output reg wr_data, OEN, CEN;
40 ////////////ROM ports//////////////////
41 input[15:0] PM_out;
42 output reg [15:0] PC;
43 ////////////////////////////////
44
45
46 //-------------------------------------------------------------
47 //-- 1 ISA Parameters
48 //-------------------------------------------------------------
49  parameter [7:0] MPY     = 8'b00000000; //1.
50  parameter [7:0] MPYK    = 8'b10100000; //2.
51  parameter [7:0] MAC     = 8'h02; //3.
52  parameter [7:0] OR      = 8'h03; //4.
53  parameter [7:0] XOR     = 8'h04; //5.
54  parameter [15:0] SPAC   = 16'h0100; //6.
55  parameter [2:0] SUB     = 3'h1; //7.
56  parameter [7:0] SUBS    = 8'h05; //8.
57  parameter [2:0] ADD     = 3'h2; //9.
58  parameter [7:0] ADDS    = 8'h06; //10.
59  parameter [7:0] AND     = 8'h07; //11.
60  parameter [15:0] BU     = 16'h010F; //12.
61  parameter [15:0] BGEZ   = 16'h0101; //14.
62  parameter [15:0] BGZ    = 16'h0102; //15.
63  parameter [15:0] BLEZ   = 16'h0103; //16.
64  parameter [15:0] BLZ    = 16'h0104; //17.
65  parameter [15:0] BNZ    = 16'h0105; //18.
66  parameter [15:0] BV     = 16'h0106; //19.
67  parameter [15:0] BZ     = 16'h0107; //20.
68  parameter [2:0] LAC     = 3'h3; //21.
```

```
69   parameter [7:0] LACK    = 8'b10100001; //22.
70   parameter [7:0] LAR     = 5'b11000; //23.
71   parameter [7:0] LARK    = 5'b11001; //24.
72   parameter [7:0] LARKH   = 5'b11011; //25.
73   parameter [7:0] LARP    = 8'b10100011; //26.
74   parameter [7:0] LDP     = 8'h09; //27.
75   parameter [7:0] LDPK    = 8'b10100100; //28.
76   parameter [7:0] LT      = 8'h0a; //29.
77   parameter [7:0] LTA     = 8'h0b; //30.
78   parameter [7:0] LTD     = 8'h0c; //31.
79   parameter [7:0] LTP     = 8'h0d; //32.
80   parameter [7:0] LTS     = 8'h0e; //33.
81   parameter [7:0] MAR     = 8'h0f; //34.
82   parameter [15:0] PAC    = 16'h011F; //35.
83   parameter [15:0] ROVM   = 16'h012F; //36.
84   parameter [2:0] SAC     = 3'h4; //37.
85   parameter [7:0] SAR     = 5'b11010; //38.
86   parameter [15:0] SOVM   = 16'h013F; //39.
87   parameter [7:0] TBLR    = 8'h11; //40.
88   parameter [7:0] TBLW    = 8'h12; //41.
89   parameter [15:0] NOP    = 16'h014F; //42.
90   parameter [15:0] ZAC    = 16'h015F; //43.
91   parameter [7:0] ZALH    = 8'h13; //44.
92   parameter [7:0] ZALS    = 8'h14; //45.
93   parameter [15:0] APAC   = 16'h016F; //46.
94   parameter [6:0] CMPSIMD = 7'b0001101; //47.
95   parameter [7:0] SUBSIMD = 8'h16; //48.
96   parameter [7:0] ADDSIMD = 8'h17; //49.
97   parameter [8:0] BANZ    = 8'h18; //13.
98   parameter [15:0] PUSH   = 16'h018F; //50.
99   parameter [15:0] POP    = 16'h017F; //51.
100  parameter [15:0] CALL   = 16'h01AF; //52.
101  parameter [15:0] RET    = 16'h019F; //53.
102
103 //-------------------------------------------------------------
104 //-- 2 Internal registers (& wires)
105 //-------------------------------------------------------------
106  reg [15:0] AR [7:0]; // 7 Auxiliary Registers of width 16-bits each
107  reg [2:0] ARP, PARP; // 2 Registers to store current and previous AR
         pointers
108
109  reg  [7:0] DPPTR;
110
111  reg [31:0] acc, Preg;
112  reg [15:0] Treg;
113
114  wire [15:0] SR_wire;
115  reg [15:0] SR;       // Status register (4*CNVZ) for SIMD instuctions
116
```

```
117   reg [4:0] SP;       // Stack pointer
118   reg [4:0] CSP;// Call stack pointer ;
119  //-----------------------------------------------------------
120  //-- 3 Pipeline registers
121  //-----------------------------------------------------------
122   reg [4:0] sreg2, sreg3, sreg4;     // Used to temporarily Shift value
        between pipeline stages
123   reg [15:0] JAddr2, JAddr3, JAddr4, JAddr;// Used to temporarily store Jump
        Address between pipeline stages
124   reg [31:0] temp_acc;
125   reg [31:0] breg;
126   reg [15:0] PAR [7:0]; // 7 Auxiliary Registers of width 16-bits each
127   reg  cnt;                  //
128   reg JFlag, JFlag_del, JFlag_c, JFlag_uc;
129   reg stall_mc1, stall_mc2, stall_mc3, stall_mc4, stall_uc;//
130   reg [15:0] IR2, IR3, IR4, IR_del;
131   reg J_detect;
132   reg [15:0]DM_Addr_reg;
133  //-----------------------------------------------------------
134  //-- 4 Memory clock setup
135  //-----------------------------------------------------------
136  // assign clk_n <=~clk;
137   reg get_DMAddr;
138
139
140   reg [31:0] stack [31:0];      // 32  stack registers
141   reg [15:0] call_stack [31:0];// 32  call stack registers
142   reg [15:0] call_SR [31:0];    // 32  call SR registers
143
144
145   reg [15:0] alu_opcode, mpy_opcode, next_opcode;
146  //-----------------------------------------------------------
147  //-----------------------------------------------------------
148  //-----------------------------------------------------------
149
150   reg DM_cnt;
151
152   wire [31:0] s1_in;
153   wire [31:0] s1_out;
154
155   wire [31:0] alu_out;
156
157   wire [15:0] mult_2;
158   wire [31:0] result;
159
160   wire [31:0] Preg_wire;
161   reg [31:0] s1_in_reg, buff;
162   reg [15:0] buff_mult_2;
163   reg updated_AR;
```

```
164
165   wire branch_predict;
166   reg check_condition;
167
168   assign branch_predict = (IR4[15:0]==BZ) ?  ((acc==0)? 1:0) :
169                           (IR4[15:0]==BV) ?  ((SR[15]==0)? 1:0) :
170                           (IR4[15:0]==BNZ) ? ((acc!=0)? 1:0) :
171                           (IR4[15:0]==BLZ) ? ((acc< 0)? 1:0) :
172                           (IR4[15:0]==BLEZ) ? ((acc<=0)? 1:0) :
173                           (IR4[15:0]==BGEZ) ? ((acc>=0)? 1:0) :
174                           (IR4[15:0]==BGZ) ? ((acc> 0)? 1:0) :
175                           ((IR4[15:8]==BANZ) && (IR4[6:0]==0))? ((AR[ARP]==0)?
                                  1:0):
176                           0;
177
178   assign DM_Addr= (get_DMAddr)? (IR2[7] == 1? AR[ARP][14:0]: {DPPTR, IR2
          [6:0]}):
179                   (updated_AR)? AR[ARP][14:0]:
180                   DM_Addr_reg;
181
182   assign mult_2= ((IR4[15:8]==MAC)||(IR4[15:8]==MPY))     ? buff_mult_2:
183                  (IR4[15:8]==MPYK)                        ? {8'd0, IR4[7:0]}:
184                  0;
185
186   assign DM_in = (IR4[15:13] == SAC)                                    ?
             acc          :
187                  (IR4[15:11] == SAR)                                    ?
                              AR[IR4[10:8]]   :
188                  32'h0;
189
190   assign s1_in = ((IR4[15:8]==MAC)||(IR4[15:0]==SPAC)||(IR4[15:0]==APAC))?
          Preg:
191                  ((IR4[15:8]==OR)||(IR4[15:8]==XOR)||(IR4[15:8]==SUBS)||(IR4
                      [15:8]==AND)||(IR4[15:8]==SUBSIMD)||(IR4[15:8]==ADDSIMD)
                      ||(IR4[15:9]==CMPSIMD)
192                  ||(IR4[15:8]==LTA)||(IR4[15:8]==LTS)||(IR4[15:13]==SUB)||(
                      IR4[15:13]==ADD)||(IR4[15:13]==LAC)||(IR4[15:13]==SAC))?
                      s1_in_reg:
193                  0;
194
195
196 //-------------------------------------------------------
197 //-- 5 Instantiation of components
198 //-------------------------------------------------------
199    multiplier m1 (.scan_in0       (scan_in0),
200                   .scan_out0      (scan_out0),
201                   .scan_en        (scan_en),
202                   .test_mode      (test_mode),
203                   .a              (Treg),
```

```
204                 .b              (mult_2),
205                 .ov             (SR[15]),
206                 .product        (Preg_wire));
207
208 // Shifts input operand
209  shifter_input s1     (.scan_in0 (scan_in0),
210                 .scan_out0      (scan_out0),
211                 .scan_en        (scan_en),
212                 .test_mode      (test_mode),
213                 .shift_in       (s1_in),
214                 .opcode         (alu_opcode),
215                 .shift_out      (s1_out));
216
217  ALU      alu1   (.scan_in0       (scan_in0),
218                 .scan_out0      (scan_out0),
219                 .scan_en        (scan_en),
220                 .test_mode      (test_mode),
221                 //
222                 .a              (acc),
223                 .b              (s1_out),
224                 .opcode         (alu_opcode),
225                 .result         (alu_out),
226                 .carry          (SR_wire[15]),
227                 .negative       (SR_wire[14]),
228                 .ov             (SR_wire[13]),
229                 .zero           (SR_wire[12]),
230                 .carry_2        (SR_wire[11]),
231                 .negative_2     (SR_wire[10]),
232                 .ov_2           (SR_wire[9]),
233                 .zero_2         (SR_wire[8]),
234                 .carry_3        (SR_wire[7]),
235                 .negative_3     (SR_wire[6]),
236                 .ov_3           (SR_wire[5]),
237                 .zero_3         (SR_wire[4]),
238                 .carry_4        (SR_wire[3]),
239                 .negative_4     (SR_wire[2]),
240                 .ov_4           (SR_wire[1]),
241                 .zero_4         (SR_wire[0])
242                 );
243
244 // Shifts output operand
245  shifter_output s2     (.scan_in0        (scan_in0),
246                 .scan_out0      (scan_out0),
247                 .scan_en        (scan_en),
248                 .test_mode      (test_mode),
249                 .shift_in       (alu_out),
250                 .opcode         (next_opcode),
251                 .shift_out      (result));
252 //---------------------------------------------------------------------
```

```
253 //-- 6 Code starts here....
254 //-------------------------------------------------------------
255  always@ (posedge clk or posedge reset)
256  begin
257         if(reset)
258         begin
259                 PC <= 16'h0;
260                 AR[0] <= 16'h0;AR[1] <= 16'h0;AR[2] <=16'h0;AR[3] <=16'h0;
261                 AR[4] <= 16'h0;AR[5] <=16'h0;AR[6] <=16'h0;AR[7] <=16'h0;
262                 stall_mc1 <=0; stall_mc2 <=1; stall_mc3 <=1; stall_mc4 <=1;
263                 IR2 <=16'h0; IR3 <=16'h0; IR4 <=16'h0; DPPTR <=8'd0;
264                 wr_data <=1'b1; //Read mode
265                 ARP <=3'd0;
266                 SP <=5'd0;
267                 CSP <=5'd0;
268                 CEN <=1'b0;OEN <=1'b0;
269                 JFlag <=1'b0; JFlag_uc <=1'b0; JFlag_c <=1'b0;
270                 Treg <=16'h0;
271                 Preg <=32'h0;
272                 acc  <=32'h0;
273                 breg <=32'h0;
274                 alu_opcode <= 16'h0;
275                 mpy_opcode <= 16'h0;
276                 next_opcode <= 16'h0;
277                 DM_Addr_reg <= 15'h0;
278                 ////////////////////
279                 updated_AR<= 0;
280                 ///////
281                 buff <= 32'h0;
282                 buff_mult_2 <= 32'h0;
283                 stall_uc <=1'h0;
284                 cnt <=0;
285                 JAddr <=16'h0; JAddr2<=16'h0; JAddr3<=16'h0; JAddr4<=16'h0;
286                 //////
287                 stack[0] <=32'h0;stack[1] <=32'h0;stack[2] <=32'h0;stack[3]
                        <=32'h0;
288                 stack[4] <=32'h0;stack[5] <=32'h0;stack[6] <=32'h0;stack[7]
                        <=32'h0;
289                 stack[8] <=32'h0;stack[9] <=32'h0;stack[10] <=32'h0;stack
                        [11] <=32'h0;
290                 stack[12] <=32'h0;stack[13] <=32'h0;stack[14] <=32'h0;stack
                        [15] <=32'h0;
291                 stack[16] <=32'h0;stack[17] <=32'h0;stack[18] <=32'h0;stack
                        [19] <=32'h0;
292                 stack[20] <=32'h0;stack[21] <=32'h0;stack[22] <=32'h0;stack
                        [23] <=32'h0;
293                 stack[24] <=32'h0;stack[25] <=32'h0;stack[26] <=32'h0;stack
                        [27] <=32'h0;
```

```
294                 stack[28] <=32'h0;stack[29] <=32'h0;stack[30] <=32'h0; stack
                        [31] <=32'h0;
295                 //////
296                 call_stack[0]  <=16'h0;call_stack[1]  <=16'h0;call_stack[2]
                        <=16'h0;call_stack[3]  <=16'h0;
297                 call_stack[4]  <=16'h0;call_stack[5]  <=16'h0;call_stack[6]
                        <=16'h0;call_stack[7]  <=16'h0;
298                 call_stack[8]  <=16'h0;call_stack[9]  <=16'h0;call_stack[10]
                        <=16'h0;call_stack[11] <=16'h0;
299                 call_stack[12] <=16'h0;call_stack[13] <=16'h0;call_stack[14]
                        <=16'h0;call_stack[15] <=16'h0;
300                 call_stack[16] <=16'h0;call_stack[17] <=16'h0;call_stack[18]
                        <=16'h0;call_stack[19] <=16'h0;
301                 call_stack[20] <=16'h0;call_stack[21] <=16'h0;call_stack[22]
                        <=16'h0;call_stack[23] <=16'h0;
302                 call_stack[24] <=16'h0;call_stack[25] <=16'h0;call_stack[26]
                        <=16'h0;call_stack[27] <=16'h0;
303                 call_stack[28] <=16'h0;call_stack[29] <=16'h0;call_stack[30]
                        <=16'h0;call_stack[31] <=16'h0;
304                 //////
305                 call_SR[0]  <=16'h0;call_SR[1]  <=16'h0;call_SR[2]  <=16'h0;
                        call_SR[3]  <=16'h0;
306                 call_SR[4]  <=16'h0;call_SR[5]  <=16'h0;call_SR[6]  <=16'h0;
                        call_SR[7]  <=16'h0;
307                 call_SR[8]  <=16'h0;call_SR[9]  <=16'h0;call_SR[10] <=16'h0;
                        call_SR[11] <=16'h0;
308                 call_SR[12] <=16'h0;call_SR[13] <=16'h0;call_SR[14] <=16'h0;
                        call_SR[15] <=16'h0;
309                 call_SR[16] <=16'h0;call_SR[17] <=16'h0;call_SR[18] <=16'h0;
                        call_SR[19] <=16'h0;
310                 call_SR[20] <=16'h0;call_SR[21] <=16'h0;call_SR[22] <=16'h0;
                        call_SR[23] <=16'h0;
311                 call_SR[24] <=16'h0;call_SR[25] <=16'h0;call_SR[26] <=16'h0;
                        call_SR[27] <=16'h0;
312                 call_SR[28] <=16'h0;call_SR[29] <=16'h0;call_SR[30] <=16'h0;
                        call_SR[31] <=16'h0;
313                 //////
314         end
315         else
316         begin
317                 DM_Addr_reg <=DM_Addr ;
318                 // Fetch Data memory Address in Fetch stage or pipeline
                        stage 2
319                 if ((PM_out[15:8]==MPY)||(PM_out[15:8]==MAC)||(PM_out
                        [15:8]==OR)||(PM_out[15:8]==XOR)||(PM_out[15:8]==SUBS)||(
                        PM_out[15:8]==ADDS)||(PM_out[15:8]==AND)
320                 ||(PM_out[15:8]==LDP)||(PM_out[15:8]==LT)||(PM_out[15:8]==
                        LTA)||(PM_out[15:8]==LTP)||(PM_out[15:8]==LTS)||(PM_out
                        [15:8]==MAR)||(PM_out[15:8]==ZALH)
```

```
321                     ||(PM_out[15:8]==ZALS)||(PM_out[15:8]==SUBSIMD)||(PM_out
                            [15:8]==ADDSIMD)||(PM_out[15:9]==CMPSIMD)||(PM_out
                            [15:11]==LAR)||(PM_out[15:11]==SAR)
322                     ||(PM_out[15:13]==SUB)||(PM_out[15:13]== ADD)||(PM_out
                            [15:13]==LAC)||(PM_out[15:13]==SAC))        get_DMAddr <=
                            1;
323                     else                        get_DMAddr <= 0;
324
325                     if (updated_AR == 1)      updated_AR <= 0;
326
327                     if (JFlag_c == 1) begin
328                             JFlag_c <=0;
329                     end
330
331                     if((wr_data ==0)&&(IR2[15:13] != SAC)&&(IR2[15:13] != SAR))
                            wr_data <=1'b1; //Read mode
332 //
    ////////////////////////////////////////////////////////////////////////////////

333 //-- 6.1 Pipeline stage 4
334                     if (stall_mc4 == 0)
335                     begin
336                             case (IR4[15:13])
337                                     ADD,SUB,LAC: begin
338                                             acc<= result;
339                                             SR <= SR_wire;
340                                     end
341                             endcase
342                             case (IR4[15:11])
343                                     LAR: AR[IR4[10:8]] <=buff[15:0];
344                             endcase
345                             case (IR4[15:9])
346                                     CMPSIMD:begin
347                                             SR<=SR_wire;
348                                             acc<=result;
349                                     end
350                             endcase
351                             case (IR4[15:8] )
352                                     ADDS, SUBS,LTA,SUBS,LTS,SUBSIMD,ADDSIMD,AND,
                                            OR,XOR:begin
353                                             SR<=SR_wire;
354                                             acc<=result;
355                                     end
356                                     MAC: begin
357                                             Preg <= Preg_wire;
358                                             SR<=SR_wire;
359                                             acc<=result;
360                                     end
361                                     MPY, MPYK: begin
```

```
362                             Preg <= Preg_wire;
363                         end
364                         LACK: begin
365                             acc    <=IR4[7:0];
366                         end
367                         LT: begin
368                             if(SR[13]==0)    Treg[15:0]  <= buff
                                    [15:0];       // if overflow is
                                    reset, then number is considered
                                    positive
369                             else             begin
                                                    // if overflow is
                                     set, then number is considered
                                    negative
370                                                 Treg[15] <= buff
                                                    [31];
371                                                 Treg[14:0] <= buff
                                                    [14:0];
372                             end
373                         end
374                         LTA,LTS: begin
375                             if(SR[13]==0)    Treg[15:0]  <= buff
                                    [15:0];       // if overflow is
                                    reset, then number is considered
                                    positive
376                             else             begin
                                                    // if overflow is
                                     set, then number is considered
                                    negative
377                                                 Treg[15] <= buff
                                                    [31];
378                                                 Treg[14:0] <= buff
                                                    [14:0];
379                             end
380                         end
381                         LTP: begin
382                             acc <= Preg;
383                             if(SR[13]==0)    Treg[15:0]  <= buff
                                    [15:0];       // if overflow is
                                    reset, then number is considered
                                    positive
384                             else             begin
                                                    // if overflow is
                                     set, then number is considered
                                    negative
385                                                 Treg[15] <= buff
                                                    [31];
386                                                 Treg[14:0] <= buff
                                                    [14:0];
```

```
387                                     end
388                             end
389                             ZALH: begin
390                                             acc <={16'd0, DM_out[15:0]};
391                             end
392                             ZALS: begin
393                                             acc <={DM_out[15:0], 16'd0};
394                             end
395                             BANZ: begin
396                                     if (AR[ARP] == 0) JFlag <=0;
397                                     else begin
398                                      JFlag <=1'b1;
399                                      JFlag_c <= 1'b1;
400                                      JAddr <=JAddr4;
401                                     end
402                             end
403                             LARP:   ARP   <=IR4[2:0];
404                             LDPK:   DPPTR <=IR4[7:0];
405                             LDP:    DPPTR <=DM_out[7:0];
406                     endcase
407                     case (IR4[15:0] )
408                             APAC, SPAC: begin
409                                     acc<=result;
410                                     SR<= SR_wire;
411                             end
412                             BGEZ: begin
413                                     if (acc >= 0)begin
414                                             JFlag_c <=1'b1;
415                                             JAddr <=JAddr4;
416                                     end
417                             end
418                             BGZ: begin
419                                     if (acc > 0) begin
420                                             JFlag_c <=1'b1;
421                                             JAddr <=JAddr4;
422                                     end
423                             end
424                             BLEZ: begin
425                                     if (acc <= 0)begin
426                                             JAddr <=JAddr4;
427                                             JFlag_c <=1'b1;


428                                     end
429                             end
430                             BLZ: begin
431                                     if (acc < 0) begin
432                                             JAddr <=JAddr4;
433                                             JFlag_c <=1'b1;
```

```
434                                     end
435                                 end
436                                 BNZ:begin
437                                     if (acc != 0)begin
438                                         JAddr <=JAddr4;
439                                         JFlag_c <=1'b1;
440                                     end
441                                 end
442                                 BV:begin
443                                     if (SR[13] == 1) begin
444                                         SR[13] <=0;
445                                         JFlag_c <=1'b1;
446                                         JAddr <=JAddr4;
447                                     end
448                                 end
449                                 BZ:begin
450                                     if (acc == 0)  begin
451                                         JFlag_c <=1'b1;
452                                         JAddr <=JAddr4;
453                                     end
454                                 end
455                                 ROVM:SR[13] <= 0;
456                                 SOVM:SR[13] <= 1;
457                                 ZAC: acc <= 32'h0;
458                                 PAC: acc <= Preg;
459                                 PUSH:begin
460                                         stack[SP] <=acc;
461                                         SP <=SP + 1'b1;
462                                 end
463                                 POP:begin
464                                         acc <=stack[SP-1'b1];
465                                         SP <=SP - 1'b1;
466                                 end
467                                 CALL:begin
468                                     call_SR[CSP] <=SR;
469                                     CSP <=CSP + 1'b1;
470                                 end
471                                 RET:begin
472                                     SR <=call_SR[CSP];
473                                 end
474                         endcase
475                 end
476 //
    ////////////////////////////////////////////////////////////////////////////////

477 //-- 6.2 Pipeline stage 3
478                 if (stall_mc3 == 0)
479                 begin
480                         case (IR3[15:0])
```

```
481                             APAC, SPAC:begin
482                                     alu_opcode <= IR3;


483                                     s1_in_reg <= Preg;
484                             end
485                     endcase
486                     case (IR3[15:9])
487                             CMPSIMD:begin
488                                     alu_opcode <= IR3;


489                                     s1_in_reg <= DM_out;
490                                     if (IR3[7] == 1)begin
491                                             PAR[ARP] <=AR[ARP];
492                                             PARP <=ARP;
493                                             updated_AR <= 1;
494                                             if (JFlag_c == 1)
                                                    begin
495                                                         ARP       <= PARP;
496                                                         AR[PARP] <= PAR[PARP
                                                                ];
497                                             end
498                                             else
                                                    begin
499                                                         case ({IR3[6],IR3
                                                                [5]})
500                                                                 2'b00: AR[
                                                                    ARP] <=AR
                                                                    [ARP];
501                                                                 2'b01: AR[
                                                                    ARP] <=AR
                                                                    [ARP] +
                                                                    1;
502                                                                 2'b10: AR[
                                                                    ARP] <=AR
                                                                    [ARP] -
                                                                    1;
503                                                         endcase
504                                                         ARP <=IR3[2:0];
505                                             end
506                                     end
507                             end
508                     endcase
509                     case (IR3[15:8])
510                             OR,XOR,SUBS,ADDS,AND,SUBSIMD,ADDSIMD:begin

511                                     alu_opcode <= IR3;
```

```
512                                     s1_in_reg <= DM_out;
513                                     if (IR3[7] == 1)begin
514                                             PAR[ARP] <=AR[ARP];
515                                             PARP <=ARP;
516                                             updated_AR <= 1;
517                                             if (JFlag_c == 1)
                                                    begin
518                                                         ARP       <= PARP;
519                                                         AR[PARP] <= PAR[PARP
                                                                ];
520                                             end
521                                             else
                                                    begin
522                                                         case ({IR3[6],IR3
                                                                [5]})
523                                                             2'b00: AR[
                                                                ARP] <=AR
                                                                [ARP];
524                                                             2'b01: AR[
                                                                ARP] <=AR
                                                                [ARP] +
                                                                1;
525                                                             2'b10: AR[
                                                                ARP] <=AR
                                                                [ARP] -
                                                                1;
526                                                         endcase
527                                                         ARP <=IR3[2:0];
528                                             end
529                                     end
530                             end
531                             LTA,LTS:        begin
532                                     buff <= DM_out;
533                                     alu_opcode <= IR3;


534                                     s1_in_reg <= Preg;
535                                     if (IR3[7] == 1)begin
536                                             PAR[ARP] <=AR[ARP];
537                                             PARP <=ARP;
538                                             updated_AR <= 1;
539                                             if (JFlag_c == 1)
                                                    begin
540                                                         ARP       <= PARP;
541                                                         AR[PARP] <= PAR[PARP
                                                                ];
542                                             end
```

```
543                                         else
                                                begin
544                                                 case ({IR3[6],IR3
                                                        [5]})
545                                                     2'b00: AR[
                                                            ARP] <=AR
                                                            [ARP];
546                                                     2'b01: AR[
                                                            ARP] <=AR
                                                            [ARP] +
                                                            1;
547                                                     2'b10: AR[
                                                            ARP] <=AR
                                                            [ARP] -
                                                            1;
548                                                 endcase
549                                                 ARP <=IR3[2:0];
550                                         end
551                                 end
552                         end
553                         LTP,MAR,ZALH,ZALS,LT,LDP:begin
554                                 buff <= DM_out;
555                                 if (IR3[7] == 1)begin
556                                         PAR[ARP] <=AR[ARP];
557                                         PARP <=ARP;
558                                         updated_AR <= 1;
559                                         if (JFlag_c == 1)
                                                begin
560                                                 ARP      <= PARP;
561                                                 AR[PARP] <= PAR[PARP
                                                        ];
562                                         end
563                                         else
                                                begin
564                                                 case ({IR3[6],IR3
                                                        [5]})
565                                                     2'b00: AR[
                                                            ARP] <=AR
                                                            [ARP];
566                                                     2'b01: AR[
                                                            ARP] <=AR
                                                            [ARP] +
                                                            1;
567                                                     2'b10: AR[
                                                            ARP] <=AR
                                                            [ARP] -
                                                            1;
568                                                 endcase
569                                                 ARP <=IR3[2:0];
```

```
570                                     end
571                                 end
572                             end
573                             MPY: begin
574                                     mpy_opcode <= IR3;
575                                     buff_mult_2 <= DM_out;
576                                     if (IR3[7] == 1)begin
577                                             PAR[ARP] <=AR[ARP];
578                                             PARP <=ARP;
579                                             updated_AR <= 1;
580                                             if (JFlag_c == 1)
                                                    begin
581                                                         ARP       <= PARP;
582                                                         AR[PARP] <= PAR[PARP
                                                                ];
583                                             end
584                                             else
                                                    begin
585                                                         case ({IR3[6],IR3
                                                                [5]})
586                                                                 2'b00: AR[
                                                                    ARP] <=AR
                                                                    [ARP];
587                                                                 2'b01: AR[
                                                                    ARP] <=AR
                                                                    [ARP] +
                                                                    1;
588                                                                 2'b10: AR[
                                                                    ARP] <=AR
                                                                    [ARP] -
                                                                    1;
589                                                         endcase
590                                                         ARP <=IR3[2:0];
591                                             end
592                                     end
593                             end
594                             MPYK: mpy_opcode <= IR3;
595                             MAC: begin
596                                     mpy_opcode <= IR3;
597                                     alu_opcode <= IR3;


598                                     s1_in_reg <= Preg;
599                                     buff_mult_2 <= DM_out;

600                                     updated_AR <= 1;
601                                     if (IR3[7] == 1)begin
602                                             PAR[ARP] <=AR[ARP];
603                                             PARP <=ARP;
```

```
604                                     if (JFlag_c == 1)
                                            begin
605                                             ARP       <= PARP;
606                                             AR[PARP] <= PAR[PARP
                                                    ];
607                                     end
608                                     else
                                            begin
609                                             case ({IR3[6],IR3
                                                    [5]})
610                                                     2'b00: AR[
                                                            ARP] <=AR
                                                            [ARP];
611                                                     2'b01: AR[
                                                            ARP] <=AR
                                                            [ARP] +
                                                            1;
612                                                     2'b10: AR[
                                                            ARP] <=AR
                                                            [ARP] -
                                                            1;
613                                             endcase
614                                             ARP <=IR3[2:0];
615                                     end
616                             end
617                     end
618                     BANZ: begin
619                             if (IR3[7] == 1)begin
620                                     PAR[ARP] <=AR[ARP];
621                                     PARP <=ARP;
622                                     updated_AR <= 1;
623                                     if (JFlag_c == 1)
                                            begin
624                                             ARP       <= PARP;
625                                             AR[PARP] <= PAR[PARP
                                                    ];
626                                     end
627                                     else
                                            begin
628                                             case ({IR3[6],IR3
                                                    [5]})
629                                                     2'b00: AR[
                                                            ARP] <=AR
                                                            [ARP];
630                                                     2'b01: AR[
                                                            ARP] <=AR
                                                            [ARP] +
                                                            1;
```

```
631                                                                 2'b10: AR[
                                                                        ARP] <=AR
                                                                        [ARP] -
                                                                        1;
632                                                         endcase
633                                                         ARP <=IR3[2:0];
634                                                 end
635                                         end
636                                 end
637                         endcase
638                         case (IR3[15:13])
639                                 SUB, ADD,LAC: begin
640                                         alu_opcode <= IR3;
641                                         updated_AR <= 1;
642                                         s1_in_reg <= buff;
643                                         if (IR3[7] == 1)begin
644                                                 PAR[ARP] <=AR[ARP];
645                                                 buff <= DM_out;
646                                                 s1_in_reg <= DM_out;

647                                                 PARP <=ARP;
648                                                 if (JFlag_c == 1)
                                                        begin
649                                                         ARP       <= PARP;
650                                                         AR[PARP] <= PAR[PARP
                                                                ];
651                                                 end
652                                                 else
                                                        begin
653                                                         case ({IR3[6],IR3
                                                                [5]})
654                                                                 2'b00: AR[
                                                                        ARP] <=AR
                                                                        [ARP];
655                                                                 2'b01: AR[
                                                                        ARP] <=AR
                                                                        [ARP] +
                                                                        1;
656                                                                 2'b10: AR[
                                                                        ARP] <=AR
                                                                        [ARP] -
                                                                        1;
657                                                         endcase
658                                                         ARP <=IR3[2:0];
659                                                 end
660                                         end
661                                 end
662                                 SAC: begin
663                                         alu_opcode <= IR3;
```

```
664                                     s1_in_reg <= DM_out;

665                                     updated_AR <= 1;
666                                     if (IR3[7] == 1)begin
667                                             PAR[ARP] <=AR[ARP];
668                                             PARP <=ARP;
669                                             if (JFlag_c == 1)
                                                    begin
670                                                         ARP       <= PARP;
671                                                         AR[PARP] <= PAR[PARP
                                                                ];
672                                             end
673                                             else
                                                    begin
674                                                         case ({IR3[6],IR3
                                                                [5]})
675                                                             2'b00: AR[
                                                                ARP] <=AR
                                                                [ARP];
676                                                             2'b01: AR[
                                                                ARP] <=AR
                                                                [ARP] +
                                                                1;
677                                                             2'b10: AR[
                                                                ARP] <=AR
                                                                [ARP] -
                                                                1;
678                                                         endcase
679                                                         ARP <=IR3[2:0];
680                                             end
681                                     end
682                             end
683             endcase
684             case (IR3[15:11])
685                     LAR: begin
686                                     buff <= DM_out;
687                                     if (IR3[7] == 1)begin
688                                             PAR[ARP] <=AR[ARP];
689                                             PARP <=ARP;
690                                             updated_AR <= 1;
691                                             if (JFlag_c == 1)
                                                    begin
692                                                         ARP       <= PARP;
693                                                         AR[PARP] <= PAR[PARP
                                                                ];
694                                             end
695                                             else
                                                    begin
```

```
696                                                 case ({IR3[6],IR3
                                                        [5]})
697                                                     2'b00: AR[
                                                            ARP] <=AR
                                                            [ARP];
698                                                     2'b01: AR[
                                                            ARP] <=AR
                                                            [ARP] +
                                                            1;
699                                                     2'b10: AR[
                                                            ARP] <=AR
                                                            [ARP] -
                                                            1;
700                                                 endcase
701                                                 ARP <=IR3[2:0];
702                                             end
703                                         end
704                                 end
705                                 SAR: begin
706                                         if (IR3[7] == 1)begin
707                                             PAR[ARP] <=AR[ARP];
708                                             PARP <=ARP;
709                                             updated_AR <= 1;
710                                             if (JFlag_c == 1)
                                                    begin
711                                                     ARP       <= PARP;
712                                                     AR[PARP] <= PAR[PARP
                                                            ];
713                                             end
714                                             else
                                                    begin
715                                                     case ({IR3[6],IR3
                                                            [5]})
716                                                         2'b00: AR[
                                                                ARP] <=AR
                                                                [ARP];
717                                                         2'b01: AR[
                                                                ARP] <=AR
                                                                [ARP] +
                                                                1;
718                                                         2'b10: AR[
                                                                ARP] <=AR
                                                                [ARP] -
                                                                1;
719                                                     endcase
720                                                     ARP <=IR3[2:0];
721                                             end
722                                         end
723                                 end
```

```
724                                     LARK: AR[IR3[10:8]][7:0]  <=IR3[7:0];
725                                     LARKH:AR[IR3[10:8]][15:8] <=IR3[7:0];
726                             endcase
727                     end
728     //-- 6.3 Pipeline stage 2
729                     if (stall_mc2 == 0)
730                     begin
731                             case (IR2[15:9] )
732                                     CMPSIMD:begin
733                                             wr_data <=1'b1; // For read
734                                     end
735                             endcase
736                             case (IR2[15:8] )
737                                     BANZ:begin
738                                             JAddr2 <=PM_out;
739                                       end
740                                     MPY,MAC,OR,XOR,SUBS,ADDS,AND,LDP,LT,LTA,LTP,
                                          LTS,MAR,ZALH,ZALS,SUBSIMD,ADDSIMD:begin
741                                             wr_data <=1'b1; // For read
742                                     end
743                             endcase
744                             case (IR2[15:11])
745                                     LAR: begin
746                                             wr_data <=1'b1; // For read
747                                     end
748                                     SAR: begin
749                                             next_opcode<= IR2;
750                                             wr_data <=1'b0; // For write
751                                     end
752                             endcase
753                             case (IR2[15:13])
754                                     SUB, ADD, LAC:begin
755                                             wr_data <=1'b1; // For read
756                                             if (IR2[7] ==0) buff = DM_out;
757                                     end
758                                     SAC: begin


759                                             next_opcode<= IR2;
760                                             wr_data <=1'b0; // For write
761                                     end

762                             endcase
763                             case (IR2[15:0] )
764                                     BGEZ,BGZ,BLEZ,BLZ,BNZ,BV,BZ:begin
765                                             if (branch_predict == 0)
                                                  JAddr <=PM_out;
766                                             else              begin
                                                  JAddr2<=PM_out; end
```

```
767                                     end
768                                     BU:   begin
769                                             if (branch_predict == 0)
                                                JAddr <=PM_out;
770                                             else            begin
                                                JAddr2<=PM_out; end
771                                     end
772                                     CALL:begin
773                                             call_stack[CSP] <=PC;
774                                             if (branch_predict == 0)
                                                JAddr <=PM_out;
775                                             else            begin
                                                JAddr2<=PM_out; end
776                                     end
777                                     RET:begin
778                                             CSP <=CSP - 1;
779                                             if (branch_predict == 0)
                                                JAddr <=call_stack[CSP-1][14:0];

780                                             else            begin
                                                JAddr2<=call_stack[CSP-1][14:0];
                                                        end
781                                     end
782                             endcase
783                     end
784 //
    ////////////////////////////////////////////////////////////////////////////////

785                     if (stall_mc3 == 0)
786                     begin
787                             if (IR4[15:0] ==BU    ||
788                             IR4[15:0] ==BGEZ ||
789                             IR4[15:0] ==BGZ   ||
790                             IR4[15:0] ==BLEZ ||
791                             IR4[15:0] ==BLZ   ||
792                             IR4[15:0] ==BNZ   ||
793                             IR4[15:0] ==BV    ||
794                             IR4[15:0] ==BZ    ||
795                             IR4[15:0] ==CALL ||
796                             IR4[15:0] ==RET   ||
797                             IR4[15:8] ==BANZ )begin
798                                             stall_mc3 <=1'b1; IR4 <=16'hffff;
799                             end
800                             else if (JFlag_c == 1)  begin
801                                     stall_mc3 <=1;IR4 <=16'hffff;
802                             end
803                             else begin
804                                     stall_mc4 <=0;
805                                     IR4<= IR3;
```

```
806                         end
807                         JAddr4<= JAddr3;
808                 end
809 //
    ////////////////////////////////////////////////////////////////////////////////

810                 if (stall_mc2 == 0)
811                 begin
812                         if (IR3[15:0] ==BU    ||
813                         IR3[15:0] ==BGEZ ||
814                         IR3[15:0] ==BGZ   ||
815                         IR3[15:0] ==BLEZ ||
816                         IR3[15:0] ==BLZ   ||
817                         IR3[15:0] ==BNZ   ||
818                         IR3[15:0] ==BV    ||
819                         IR3[15:0] ==BZ    ||
820                         IR3[15:0] ==CALL ||
821                         IR3[15:0] ==RET   ||
822                         IR3[15:8] ==BANZ)begin  stall_mc2 <=1;  IR3 <=16'
                            hffff; end
823                         else if (JFlag_c == 1)  begin
824                                 stall_mc2 <=1;IR3 <=16'hffff;
825                         end
826                         else begin
                               stall_mc3 <=0;  IR3 <=IR2;       end
827                         JAddr3 <=JAddr2;
828                 end
829 //
    ////////////////////////////////////////////////////////////////////////////////

830                 if (stall_mc1 == 0)
831                 begin
832                         if(check_condition) begin
833                                 JAddr <=JAddr2;
834                                 check_condition <= 0;
835                         end
836
837                         if (branch_predict)                                PC
                           <=JAddr;
838                         else if ((IR2[13:0]==BU)||(IR2[15:0]==CALL))       PC<=
                             PM_out;
839                         else if (IR2[15:0]==RET)                           PC<=
                             call_stack[CSP-1][14:0] + 2;
840                         else                                               PC
                           <=PC + 1'b1;
841
842                         if (     IR2[15:0] ==BU    ||
843                         IR2[15:0] ==BGEZ ||
844                         IR2[15:0] ==BGZ   ||
```

```
845                             IR2[15:0] ==BLEZ ||
846                             IR2[15:0] ==BLZ  ||
847                             IR2[15:0] ==BNZ  ||
848                             IR2[15:0] ==BZ   ||
849                             IR2[15:0] ==BV   ||
850                             IR2[15:0] ==CALL ||
851                             IR2[15:0] ==RET  ||
852                             IR2[15:8] ==BANZ )                                  IR2
                                    <=16'hffff;
853                             else begin
                                    stall_mc2 <=0;  IR2 <=PM_out;    end
854                     end
855 //
    ////////////////////////////////////////////////////////////////////////////////

856 //
    ////////////////////////////////////////////////////////////////////////////////

857                     if (IR2 == 16'hffff) stall_mc1 <=0;
858 //
    ////////////////////////////////////////////////////////////////////////////////

859             end
860  end
861
862 endmodule
```

I.1.2 ALU

```
1  //
       ////////////////////////////////////////////////////////////////////////////////
2  // Author         : Shashank Simha
3  // Date           : 12/12/2017
4  // University     : Rochester Institute of Technology
5  // Description    : This is a part of the DSP implemented for grad project
6  //
       ////////////////////////////////////////////////////////////////////////////////
7  module ALU (
8          reset,
9          clk,
10         scan_in0,
11         scan_en,
12         test_mode,
13         scan_out0,
14         a,
15         b,
16         opcode,
17         result,
18         ov,
19         carry,
20         negative,
21         zero,
22         ov_2,
23         carry_2,
24         negative_2,
25         zero_2,
26         ov_3,
27         carry_3,
28         negative_3,
29         zero_3,
30         ov_4,
31         carry_4,
32         negative_4,
33         zero_4   );
34
35 input
36     reset,                      // system reset
37     clk;                        // system clock
38
39 input
40     scan_in0,                   // test scan mode data input
41     scan_en,                    // test scan mode enable
42     test_mode;                  // test mode select
```

```
43
44  output
45      scan_out0;                    // test scan mode data output
46
47  input [31:0]a,b;
48  input [15:0]opcode;
49  output[31:0]result;
50  output  ov,carry,negative,zero,ov_2,carry_2,negative_2,zero_2,ov_3,carry_3,
        negative_3,zero_3,ov_4,carry_4,negative_4,zero_4;
51  //-------------------------------------------------------------
52  //-- 1 ISA Parameters
53  //-------------------------------------------------------------
54   parameter [7:0] MPY      = 8'b00000000; //1.
55   parameter [7:0] MPYK     = 8'b10100000; //2.
56   parameter [7:0] MAC      = 8'h02; //3.
57   parameter [7:0] OR       = 8'h03; //4.
58   parameter [7:0] XOR      = 8'h04; //5.
59   parameter [15:0] SPAC    = 16'h0100; //6.
60   parameter [2:0] SUB      = 3'h1; //7.
61   parameter [7:0] SUBS     = 8'h05; //8.
62   parameter [2:0] ADD      = 3'h2; //9.
63   parameter [7:0] ADDS     = 8'h06; //10.
64   parameter [7:0] AND      = 8'h07; //11.
65   parameter [15:0] BU      = 16'h010F; //12.
66   parameter [15:0] BGEZ    = 16'h0101; //14.
67   parameter [15:0] BGZ     = 16'h0102; //15.
68   parameter [15:0] BLEZ    = 16'h0103; //16.
69   parameter [15:0] BLZ     = 16'h0104; //17.
70   parameter [15:0] BNZ     = 16'h0105; //18.
71   parameter [15:0] BV      = 16'h0106; //19.
72   parameter [15:0] BZ      = 16'h0107; //20.
73   parameter [2:0] LAC      = 3'h3; //21.
74   parameter [7:0] LACK     = 8'b10100001; //22.
75   parameter [7:0] LAR      = 5'b11000; //23.
76   parameter [7:0] LARK     = 5'b11001; //24.
77   parameter [7:0] LARKH    = 5'b11011; //25.
78   parameter [7:0] LARP     = 8'b10100011; //26.
79   parameter [7:0] LDP      = 8'h09; //27.
80   parameter [7:0] LDPK     = 8'b10100100; //28.
81   parameter [7:0] LT       = 8'h0a; //29.
82   parameter [7:0] LTA      = 8'h0b; //30.
83   parameter [7:0] LTD      = 8'h0c; //31.
84   parameter [7:0] LTP      = 8'h0d; //32.
85   parameter [7:0] LTS      = 8'h0e; //33.
86   parameter [7:0] MAR      = 8'h0f; //34.
87   parameter [15:0] PAC     = 16'h011F; //35.
88   parameter [15:0] ROVM    = 16'h012F; //36.
89   parameter [2:0] SAC      = 3'h4; //37.
90   parameter [7:0] SAR      = 5'b11010; //38.
```

```
 91   parameter [15:0] NOP    = 16'h014F; //42.
 92   parameter [15:0] ZAC    = 16'h015F; //43.
 93   parameter [7:0] ZALH    = 8'h13; //44.
 94   parameter [7:0] ZALS    = 8'h14; //45.
 95   parameter [15:0] APAC   = 16'h016F; //46.
 96   parameter [6:0] CMPSIMD = 7'b0001101; //47.
 97   parameter [7:0] SUBSIMD = 8'h16; //48.
 98   parameter [7:0] ADDSIMD = 8'h17; //49.
 99   parameter [8:0] BANZ    = 8'h18; //13.
100   parameter [15:0] PUSH   = 16'h017F; //50.
101   parameter [15:0] POP    = 16'h018F; //51.
102   parameter [15:0] CALL   = 16'h01AF; //52.
103   parameter [15:0] RET    = 16'h019F; //53.
104
105  wire [31:0] out_comp;
106  wire comp_en;
107  wire [31:0] b_sub;
108  wire [7:0] a1, a2, a3, a4;
109  wire [7:0] b1, b2, b3, b4;
110  wire [3:0] Cin;
111  wire [3:0] Cout;
112  wire [7:0] S1, S2, S3, S4;
113
114  assign ov    =((opcode[15:13]==ADD)||(opcode[15:8]==ADDS)||(opcode[15:0]==
        APAC)||(opcode[15:8]==LTA)||(opcode[15:8]==MAC)||(opcode[15:8]==ADDSIMD))
              ? ((a[31] ~^ b[31])&& result[31]):
115                 ((opcode[15:0]==SPAC)||(opcode[15:8]==LTS)||(opcode[15:8]==
                       SUBSIMD))
                                                ? ((a[31]  ^ b[31])&& result[31])
116                 :0;
117  assign ov_2 =(opcode[15:8]==ADDSIMD)?  ((a[23] ~^ b[23])&& result[23]):(
        opcode[15:8]==SUBSIMD) ? ((a[23]  ^ b[23])&& result[23])        :0;
118  assign ov_3 =(opcode[15:8]==ADDSIMD)?  ((a[15] ~^ b[15])&& result[15]):(
        opcode[15:8]==SUBSIMD) ? ((a[15]  ^ b[15])&& result[15])        :0;
119  assign ov_4 =(opcode[15:8]==ADDSIMD)?  ((a[7] ~^ b[7])&& result[7]      ):(
        opcode[15:8]==SUBSIMD) ? ((a[7]  ^ b[7])&& result[7])        :0;
120
121  assign carry = Cout[0];
122  assign carry_2= Cout[1];
123  assign carry_3 = Cout[2];
124  assign carry_4 = Cout[3];
125
126  assign negative = result[31];
127  assign negative_2       = result[23];
128  assign negative_3       = result[15];
129  assign negative_4       = result[7];
130
131  assign zero             =((opcode[15:8]==ADDSIMD)||(opcode[15:8]==SUBSIMD))
              ? ((result[31:24] == 0)? 1:0):  (result==0)? 1:          0;
```

```
132 assign zero_2    = (result[23:16]==0)? 1:0;
133 assign zero_3    = (result[15:8]==0)? 1:0;
134 assign zero_4    = (result[7:0]==0)? 1:0;
135
136 assign result = ((opcode[15:13] == ADD)|||(opcode[15:8] == ADDS)|||(opcode
        [15:0] == APAC)|||(opcode[15:8] == MAC)|||(opcode[15:8] == LTA)|||(opcode
        [15:13] == SUB)||
137          (opcode[15:0] == SPAC)|||(opcode[15:8] == SUBS)|||(opcode[15:8] == LTS
                )|||(opcode[15:8] == SUBSIMD)|||(opcode[15:8] == ADDSIMD))     ? {S4
                ,S3,S2,S1}:
138                              (opcode[15:9] == CMPSIMD)

                                                    ? out_comp:
139                              (opcode[15:8] == OR)

                                                    ? (a | b):
140                              (opcode[15:8] == AND)

                                                    ? (a & b):
141                              (opcode[15:8] == XOR)

                                                    ? (a ^ b):
142                              ((opcode[15:13] == SAC)|| (opcode[15:13] == LAC))

                                                    ? b      :0;
143
144 //
        ////////////////////////////////////////////////////////////////////////////////
145 assign comp_en =  (opcode[15:9] == CMPSIMD)? 1:0;
146
147 assign a4 = a[31:24];
148 assign a3 = a[23:16];
149 assign a2 = a[15:8];
150 assign a1 = a[7:0];
151
152 assign b_sub = (opcode[15:8] == SUBSIMD)? {(~b[31:24])+1,(~b[23:16])+1,(~b
        [15:8])+1,(~b[7:0])+1}          :(~b) + 1; // 2's complement for
        subtraction
153
154 assign b4 = ((opcode[15:13] == SUB)|||(opcode[15:8] == SUBS)|||(opcode[15:8]
        == SUBSIMD)|||(opcode[15:0] == SPAC)|||(opcode[15:8] == LTS))
                          ? b_sub[31:24]:b[31:24];
155 assign b3 = ((opcode[15:13] == SUB)|||(opcode[15:8] == SUBS)|||(opcode[15:8]
        == SUBSIMD)|||(opcode[15:0] == SPAC)|||(opcode[15:8] == LTS))
                          ? b_sub[23:16]:b[23:16];
156 assign b2 = ((opcode[15:13] == SUB)|||(opcode[15:8] == SUBS)|||(opcode[15:8]
        == SUBSIMD)|||(opcode[15:0] == SPAC)|||(opcode[15:8] == LTS))
                          ? b_sub[15:8] :b[15:8] ;
```

```
157 assign b1 = ((opcode[15:13] == SUB)|||(opcode[15:8] == SUBS)|||(opcode[15:8]
      == SUBSIMD)|||(opcode[15:0] == SPAC)|||(opcode[15:8] == LTS))
                        ? b_sub[7:0]   :b[7:0];
158
159
160 assign Cin[0] = 0;
161 assign Cin[1]= ((opcode[15:8] == ADDSIMD)|||(opcode[15:8] == SUBSIMD))? 0:
              Cout[0];
162 assign Cin[2]= ((opcode[15:8] == ADDSIMD)|||(opcode[15:8] == SUBSIMD))? 0:
              Cout[1];
163 assign Cin[3]= ((opcode[15:8] == ADDSIMD)|||(opcode[15:8] == SUBSIMD))? 0:
              Cout[2];
164
165
166 adder A1 (.A              (a1),
167         .B                (b1),
168         .Cin              (Cin[0]),
169         .Cout             (Cout[0]),
170         .Sum              (S1),
171         .scan_en          (scan_en),
172         .scan_in0         (scan_in0),
173         .test_mode        (test_mode),
174         .scan_out0        (scan_out0)
175         );
176
177 adder A2 (.A              (a2),
178         .B                (b2),
179         .Cin              (Cin[1]),
180         .Cout             (Cout[1]),
181         .Sum              (S2),
182         .scan_en          (scan_en),
183         .scan_in0         (scan_in0),
184         .test_mode        (test_mode),
185         .scan_out0        (scan_out0)
186         );
187
188 adder A3 (.A              (a3),
189         .B                (b3),
190         .Cin              (Cin[2]),
191         .Cout             (Cout[2]),
192         .Sum              (S3),
193         .scan_en          (scan_en),
194         .scan_in0         (scan_in0),
195         .test_mode        (test_mode),
196         .scan_out0        (scan_out0)
197         );
198
199 adder A4 (.A              (a4),
200         .B                (b4),
```

```
201           .Cin              (Cin[3]),
202           .Cout             (Cout[3]),
203           .Sum              (S4),
204           .scan_en          (scan_en),
205           .scan_in0         (scan_in0),
206           .test_mode        (test_mode),
207           .scan_out0        (scan_out0)
208           );
209
210   compare_select comp_4 (.scan_in0(scan_in0),
211           .scan_en(scan_en),
212           .test_mode(test_mode),
213           .scan_out0(scan_out0),
214           .A(a[31:24]),
215           .B(b[31:24]),
216           .flag(opcode[8]),
217           .C(out_comp[31:24]),
218           .en(comp_en)
219           );
220   compare_select comp_3 (.scan_in0(scan_in0),
221           .scan_en(scan_en),
222           .test_mode(test_mode),
223           .scan_out0(scan_out0),
224           .A(a[23:16]),
225           .B(b[23:16]),
226           .flag(opcode[8]),
227           .C(out_comp[23:16]),
228           .en(comp_en)
229           );
230   compare_select comp_2 (.scan_in0(scan_in0),
231           .scan_en(scan_en),
232           .test_mode(test_mode),
233           .scan_out0(scan_out0),
234           .A(a[15:8]),
235           .B(b[15:8]),
236           .flag(opcode[8]),
237           .C(out_comp[15:8]),
238           .en(comp_en)
239           );
240   compare_select comp_1 (.scan_in0(scan_in0),
241           .scan_en(scan_en),
242           .test_mode(test_mode),
243           .scan_out0(scan_out0),
244           .A(a[7:0]),
245           .B(b[7:0]),
246           .flag(opcode[8]),
247           .C(out_comp[7:0]),
248           .en(comp_en)
249           );
```

```
250
251 endmodule
```

I.1.3 Input shifter

```
1  //
       ////////////////////////////////////////////////////////////////////////////////
2  // Author         : Shashank Simha
3  // Date           : 12/12/2017
4  // University     : Rochester Institute of Technology
5  // Description    : This is a part of the DSP implemented for grad project
6  //
       ////////////////////////////////////////////////////////////////////////////////
7   module shifter_input (scan_in0,
8          scan_out0,
9          scan_en,
10         test_mode,
11         shift_in,
12         opcode,
13         shift_out);
14 //-- 1 ISA Parameters
15 //-------------------------------------------------------------
16  parameter [7:0] MPY      = 8'b00000000; //1.
17  parameter [7:0] MPYK     = 8'b10100000; //2.
18  parameter [7:0] MAC      = 8'h02; //3.
19  parameter [7:0] OR       = 8'h03; //4.
20  parameter [7:0] XOR      = 8'h04; //5.
21  parameter [15:0] SPAC    = 16'h0100; //6.
22  parameter [2:0] SUB      = 3'h1; //7.
23  parameter [7:0] SUBS     = 8'h05; //8.
24  parameter [2:0] ADD      = 3'h2; //9.
25  parameter [7:0] ADDS     = 8'h06; //10.
26  parameter [7:0] AND      = 8'h07; //11.
27  parameter [15:0] BU      = 16'h010F; //12.
28  parameter [8:0] BANZ     = 8'h01; //13.
29  parameter [15:0] BGEZ    = 16'h0101; //14.
30  parameter [15:0] BGZ     = 16'h0102; //15.
31  parameter [15:0] BLEZ    = 16'h0103; //16.
32  parameter [15:0] BLZ     = 16'h0104; //17.
33  parameter [15:0] BNZ     = 16'h0105; //18.
34  parameter [15:0] BV      = 16'h0106; //19.
35  parameter [15:0] BZ      = 16'h0107; //20.
36  parameter [2:0] LAC      = 3'h3; //21.
37  parameter [7:0] LACK     = 8'b10100001; //22.
38  parameter [7:0] LAR      = 5'b11000; //23.
39  parameter [7:0] LARK     = 5'b11001; //24.
40  parameter [7:0] LARKH    = 5'b11011; //25.
41  parameter [7:0] LARP     = 8'b10100011; //26.
42  parameter [7:0] LDP      = 8'h09; //27.
```

```
43   parameter [7:0] LDPK     = 8'b10100100; //28.
44   parameter [7:0] LT       = 8'h0a; //29.
45   parameter [7:0] LTA      = 8'h0b; //30.
46   parameter [7:0] LTD      = 8'h0c; //31.
47   parameter [7:0] LTP      = 8'h0d; //32.
48   parameter [7:0] LTS      = 8'h0e; //33.
49   parameter [7:0] MAR      = 8'h0f; //34.
50   parameter [15:0] PAC     = 16'h011F; //35.
51   parameter [15:0] ROVM    = 16'h012F; //36.
52   parameter [2:0] SAC      = 3'h4; //37.
53   parameter [7:0] SAR      = 5'b11010; //38.
54   parameter [15:0] SOVM    = 16'h013F; //39.
55   parameter [7:0] TBLR     = 8'h11; //40.
56   parameter [7:0] TBLW     = 8'h12; //41.
57   parameter [15:0] NOP     = 16'h014F; //42.
58   parameter [15:0] ZAC     = 16'h015F; //43.
59   parameter [7:0] ZALH     = 8'h13; //44.
60   parameter [7:0] ZALS     = 8'h14; //45.
61   parameter [15:0] APAC    = 16'h016F; //46.
62   parameter [7:0] CMPSIMD  = 8'h15; //47.
63   parameter [7:0] SUBSIMD  = 8'h16; //48.
64   parameter [7:0] ADDSIMD  = 8'h17; //49.
65   parameter [15:0] PUSH    = 16'h017F; //50.
66   parameter [15:0] POP     = 16'h018F; //51.
67   parameter [15:0] CALL    = 16'h01AF; //52.
68   parameter [15:0] RET     = 16'h019F; //53.
69
70 input    scan_in0,
71          scan_en,
72          test_mode;
73
74 output   scan_out0;
75
76 input [31:0]shift_in;
77 input [15:0]opcode;
78
79 output [31:0]shift_out;
80
81 // Shifts operand for ADD and SUB
82 assign shift_out =  ((opcode[15:13]==ADD) || (opcode[15:13]==SUB)) ?
83                  ((opcode[12:8]==5'b00000)       ? shift_in                :
84                  (opcode[12:8]==5'b00001)        ? {1'h0,shift_in[31:1]}   :
85                  (opcode[12:8]==5'b00010)        ? {2'h0,shift_in[31:2]}   :
86                  (opcode[12:8]==5'b00011)        ? {3'h0,shift_in[31:3]}   :
87                  (opcode[12:8]==5'b00100)        ? {4'h0,shift_in[31:4]}   :
88                  (opcode[12:8]==5'b00101)        ? {5'h0,shift_in[31:5]}   :
89                  (opcode[12:8]==5'b00110)        ? {6'h0,shift_in[31:6]}   :
90                  (opcode[12:8]==5'b00111)        ? {7'h0,shift_in[31:7]}   :
91                  (opcode[12:8]==5'b01000)        ? {8'h0,shift_in[31:8]}   :
```

```
92                  (opcode[12:8]==5'b01001)        ? {9'h0,shift_in[31:9]}   :
93                  (opcode[12:8]==5'b01010)        ? {10'h0,shift_in[31:10]}:
94                  (opcode[12:8]==5'b01011)        ? {11'h0,shift_in[31:11]}:
95                  (opcode[12:8]==5'b01100)        ? {12'h0,shift_in[31:12]}:
96                  (opcode[12:8]==5'b01101)        ? {13'h0,shift_in[31:13]}:
97                  (opcode[12:8]==5'b01110)        ? {14'h0,shift_in[31:14]}:
98                  (opcode[12:8]==5'b01111)        ? {15'h0,shift_in[31:15]}:
99                  (opcode[12:8]==5'b10000)        ? {16'h0,shift_in[31:16]}:
100                 (opcode[12:8]==5'b10001)        ? {17'h0,shift_in[31:17]}:
101                 (opcode[12:8]==5'b10010)        ? {18'h0,shift_in[31:18]}:
102                 (opcode[12:8]==5'b10011)        ? {19'h0,shift_in[31:19]}:
103                 (opcode[12:8]==5'b10100)        ? {20'h0,shift_in[31:20]}:
104                 (opcode[12:8]==5'b10101)        ? {21'h0,shift_in[31:21]}:
105                 (opcode[12:8]==5'b10110)        ? {22'h0,shift_in[31:22]}:
106                 (opcode[12:8]==5'b10111)        ? {23'h0,shift_in[31:23]}:
107                 (opcode[12:8]==5'b11000)        ? {24'h0,shift_in[31:24]}:
108                 (opcode[12:8]==5'b11001)        ? {25'h0,shift_in[31:25]}:
109                 (opcode[12:8]==5'b11010)        ? {26'h0,shift_in[31:26]}:
110                 (opcode[12:8]==5'b11011)        ? {27'h0,shift_in[31:27]}:
111                 (opcode[12:8]==5'b11100)        ? {28'h0,shift_in[31:28]}:
112                 (opcode[12:8]==5'b11101)        ? {29'h0,shift_in[31:29]}:
113                 (opcode[12:8]==5'b11110)        ? {30'h0,shift_in[31:30]}:
114                 (opcode[12:8]==5'b11111)        ? {31'h0,shift_in[31]}    :
115                 32'h0):
116                 shift_in;
117
118 endmodule
```

I.1.4 Output shifter

```
1  //
       ////////////////////////////////////////////////////////////////////////////////
2  // Author         : Shashank Simha
3  // Date           : 12/12/2017
4  // University     : Rochester Institute of Technology
5  // Description    : This is a part of the DSP implemented for grad project
6  //
       ////////////////////////////////////////////////////////////////////////////////
7   module shifter_output (scan_in0,
8          scan_out0,
9          scan_en,
10         test_mode,
11         shift_in,
12         opcode,
13         shift_out);
14 //-- 1 ISA Parameters
15 //-----------------------------------------------------------
16  parameter [7:0] MPY      = 8'b00000000; //1.
17  parameter [7:0] MPYK     = 8'b10100000; //2.
18  parameter [7:0] MAC      = 8'h02; //3.
19  parameter [7:0] OR       = 8'h03; //4.
20  parameter [7:0] XOR      = 8'h04; //5.
21  parameter [15:0] SPAC    = 16'h0100; //6.
22  parameter [2:0] SUB      = 3'h1; //7.
23  parameter [7:0] SUBS     = 8'h05; //8.
24  parameter [2:0] ADD      = 3'h2; //9.
25  parameter [7:0] ADDS     = 8'h06; //10.
26  parameter [7:0] AND      = 8'h07; //11.
27  parameter [15:0] BU      = 16'h010F; //12.
28  parameter [8:0] BANZ     = 8'h01; //13.
29  parameter [15:0] BGEZ    = 16'h0101; //14.
30  parameter [15:0] BGZ     = 16'h0102; //15.
31  parameter [15:0] BLEZ    = 16'h0103; //16.
32  parameter [15:0] BLZ     = 16'h0104; //17.
33  parameter [15:0] BNZ     = 16'h0105; //18.
34  parameter [15:0] BV      = 16'h0106; //19.
35  parameter [15:0] BZ      = 16'h0107; //20.
36  parameter [2:0] LAC      = 3'h3; //21.
37  parameter [7:0] LACK     = 8'b10100001; //22.
38  parameter [7:0] LAR      = 5'b11000; //23.
39  parameter [7:0] LARK     = 5'b11001; //24.
40  parameter [7:0] LARKH    = 5'b11011; //25.
41  parameter [7:0] LARP     = 8'b10100011; //26.
42  parameter [7:0] LDP      = 8'h09; //27.
```

```
43   parameter [7:0] LDPK     = 8'b10100100; //28.
44   parameter [7:0] LT       = 8'h0a; //29.
45   parameter [7:0] LTA      = 8'h0b; //30.
46   parameter [7:0] LTD      = 8'h0c; //31.
47   parameter [7:0] LTP      = 8'h0d; //32.
48   parameter [7:0] LTS      = 8'h0e; //33.
49   parameter [7:0] MAR      = 8'h0f; //34.
50   parameter [15:0] PAC     = 16'h011F; //35.
51   parameter [15:0] ROVM    = 16'h012F; //36.
52   parameter [2:0] SAC      = 3'h4; //37.
53   parameter [7:0] SAR      = 5'b11010; //38.
54   parameter [15:0] SOVM    = 16'h013F; //39.
55   parameter [7:0] TBLR     = 8'h11; //40.
56   parameter [7:0] TBLW     = 8'h12; //41.
57   parameter [15:0] NOP     = 16'h014F; //42.
58   parameter [15:0] ZAC     = 16'h015F; //43.
59   parameter [7:0] ZALH     = 8'h13; //44.
60   parameter [7:0] ZALS     = 8'h14; //45.
61   parameter [15:0] APAC    = 16'h016F; //46.
62   parameter [7:0] CMPSIMD = 8'h15; //47.
63   parameter [7:0] SUBSIMD = 8'h16; //48.
64   parameter [7:0] ADDSIMD = 8'h17; //49.
65   parameter [15:0] PUSH    = 16'h017F; //50.
66   parameter [15:0] POP     = 16'h018F; //51.
67   parameter [15:0] CALL    = 16'h01AF; //52.
68   parameter [15:0] RET     = 16'h019F; //53.
69
70 input    scan_in0,
71          scan_en,
72          test_mode;
73
74 output   scan_out0;
75
76 input [31:0]shift_in;
77 input [15:0]opcode;
78
79 output [31:0]shift_out;
80
81 // Shifts output for SAC
82 assign shift_out =  ((opcode[15:13]==SAC) || (opcode[15:13]==LAC)) ?
83                  ((opcode[12:8]==5'b00000)       ? shift_in                :
84                  (opcode[12:8]==5'b00001)        ? {1'h0,shift_in[31:1]}   :
85                  (opcode[12:8]==5'b00010)        ? {2'h0,shift_in[31:2]}   :
86                  (opcode[12:8]==5'b00011)        ? {3'h0,shift_in[31:3]}   :
87                  (opcode[12:8]==5'b00100)        ? {4'h0,shift_in[31:4]}   :
88                  (opcode[12:8]==5'b00101)        ? {5'h0,shift_in[31:5]}   :
89                  (opcode[12:8]==5'b00110)        ? {6'h0,shift_in[31:6]}   :
90                  (opcode[12:8]==5'b00111)        ? {7'h0,shift_in[31:7]}   :
91                  (opcode[12:8]==5'b01000)        ? {8'h0,shift_in[31:8]}   :
```

```
92                  (opcode[12:8]==5'b01001)        ? {9'h0,shift_in[31:9]}   :
93                  (opcode[12:8]==5'b01010)        ? {10'h0,shift_in[31:10]}:
94                  (opcode[12:8]==5'b01011)        ? {11'h0,shift_in[31:11]}:
95                  (opcode[12:8]==5'b01100)        ? {12'h0,shift_in[31:12]}:
96                  (opcode[12:8]==5'b01101)        ? {13'h0,shift_in[31:13]}:
97                  (opcode[12:8]==5'b01110)        ? {14'h0,shift_in[31:14]}:
98                  (opcode[12:8]==5'b01111)        ? {15'h0,shift_in[31:15]}:
99                  (opcode[12:8]==5'b10000)        ? {16'h0,shift_in[31:16]}:
100                 (opcode[12:8]==5'b10001)        ? {17'h0,shift_in[31:17]}:
101                 (opcode[12:8]==5'b10010)        ? {18'h0,shift_in[31:18]}:
102                 (opcode[12:8]==5'b10011)        ? {19'h0,shift_in[31:19]}:
103                 (opcode[12:8]==5'b10100)        ? {20'h0,shift_in[31:20]}:
104                 (opcode[12:8]==5'b10101)        ? {21'h0,shift_in[31:21]}:
105                 (opcode[12:8]==5'b10110)        ? {22'h0,shift_in[31:22]}:
106                 (opcode[12:8]==5'b10111)        ? {23'h0,shift_in[31:23]}:
107                 (opcode[12:8]==5'b11000)        ? {24'h0,shift_in[31:24]}:
108                 (opcode[12:8]==5'b11001)        ? {25'h0,shift_in[31:25]}:
109                 (opcode[12:8]==5'b11010)        ? {26'h0,shift_in[31:26]}:
110                 (opcode[12:8]==5'b11011)        ? {27'h0,shift_in[31:27]}:
111                 (opcode[12:8]==5'b11100)        ? {28'h0,shift_in[31:28]}:
112                 (opcode[12:8]==5'b11101)        ? {29'h0,shift_in[31:29]}:
113                 (opcode[12:8]==5'b11110)        ? {30'h0,shift_in[31:30]}:
114                 (opcode[12:8]==5'b11111)        ? {31'h0,shift_in[31]}    :
115                 32'h0):
116                 shift_in;
117 endmodule
```

I.1.5 Compare select unit

```
1  //////////////////////////////////////////////////////////////////////////////////
2  // Author        : Shashank Simha
3  // Date          : 12/12/2017
4  // University    : Rochester Institute of Technology
5  // Description   : This is a part of the DSP implemented for grad project
6  //////////////////////////////////////////////////////////////////////////////////
7  module compare_select(scan_in0, scan_en, test_mode, scan_out0, A,B,flag,C,
       en );
8
9  input scan_in0, scan_en, test_mode;
10 output scan_out0;
11
12 input [7:0] A,B;
13 input flag, en;
14
15 output [7:0] C;
16
17 assign C = en ? (flag? ((A>B)? A : B):((A>B)? B : A) ):0; //if flag ==1 -> C
       = Greater
18
19 endmodule
```

I.1.6 Multiplier

```
1  module   multiplier (scan_in0,  scan_out0,scan_en,test_mode,a,  b, ov,
       product);
2
3  input   scan_in0, scan_en,test_mode;
4  output  scan_out0;
5  input [15:0]a, b;
6  output[31:0]product;
7  input ov;
8
9
10 wire [15:0] abs_a, abs_b;
11 wire [15:0] twos_comp_a, twos_comp_b;
12
13 parameter PSAT = 32'h7fffffff;
14 parameter NSAT = 32'h80000000;
15
16 wire [31:0] abs_result;
17
18 assign twos_comp_a = ((~a) + 1) ;
19 assign twos_comp_b = ((~b) + 1) ;
20
21 assign abs_a = a[15] ?
22        (ov ? ((a == NSAT) ? PSAT : twos_comp_a) : twos_comp_a) : a ;
23 assign abs_b = b[15] ?
24        (ov ? ((b == NSAT) ? PSAT : twos_comp_b) : twos_comp_b) : b ;
25 assign abs_result = abs_a * abs_b ;
26 assign product = (a[15] ^ b[15]) ? ((~abs_result) + 1) : abs_result ;
27
28 endmodule
```

I.1.7 Adder

```
1  //
       ////////////////////////////////////////////////////////////////////////////////
2  // Author         : Shashank Simha
3  // Date           : 12/12/2017
4  // University     : Rochester Institute of Technology
5  // Description    : This is a part of the DSP implemented for grad project
6  //
       ////////////////////////////////////////////////////////////////////////////////
7  module adder (scan_in0, scan_en, test_mode, scan_out0, A,B,Cin,Cout,Sum);
8
9  input scan_in0, scan_en, test_mode;
10 output scan_out0;
11
12 input [7:0] A,B;
13 input Cin;
14 output [7:0] Sum;
15 output Cout;
16
17 wire [7:0] G;
18 wire [7:0] P;
19 wire [7:0] C;
20
21 assign {Cout,Sum}= A+B+Cin;
22
23 endmodule
```

I.2 Assembler designed in Perl

```
1  ##############################################################################
2  # Author        : Shashank Simha
3  # Date          : 12/12/2017
4  # University    : Rochester Institute of Technology
5  # Description   : This is a part of the ASSEMBLER for DSP implemented
6  #                 for grad project
7  ##############################################################################
8  # use strict;
9  # use warnings;
10
11 my $name = "median_filter_first_try";
12
13 my $assembly_file= $name.".txt";
14 my $mif_file     = $name.".mif";
15 my $hex_file     = $name.".hex";
16
17 my @code;
18 my @comments;
19
20 my @line_number;
21 my $line_count;
22 my $error_count;
23
24 my @code_rearr;
25
26 my @opcode_extracted;
27 my @non_opcode_extracted;
28 my %jmp_in_label_extracted;
29 my %jmp_out_label_extracted;
30 my @jmp_labels_called;
31 my $j =0 ;
32
33 my @opcode_generated;
34 my @non_opcode_generated;
35 my @jmp_addr_generated;
36
37 my @jmp_addr_generated_nonconverted;
38
39
40 my $mode;
41 my $dir_addr;
42 my $ar;
43 my $narp;
44 my $constant;
45 my $incr_oper;
46 my $shift;
```

```
47 ######################
48 my $greater_lesser;
49 ######################
50
51 my $jmp_label;
52
53
54 my %add_sub_ar= (
55           "*"  => "00",
56           "*+" => "01",
57           "*-" => "10"
58 );
59
60 my %opcode_3bit = (
61    "SUB" => "001",
62    "ADD" => "010",
63    "LAC" => "011",
64    "SAC" => "100"
65   ) ;
66
67  my %opcode_5bit = (
68    "LARKH"    => "11011",
69    "LARK"     => "11001",
70    "LAR"      => "11000",
71    "SAR"      => "11010"
72   );
73
74  my %opcode_7bit = (
75    "CMPSIMD" => "0001101",
76   );
77  my %opcode_8bit = (
78    #KEY       =>  01234567
79    "MPY"      => "00000000",
80    "MPYK"     => "10100000",
81    "MAC"      => "00000010",
82    "OR"       => "00000011",
83    "XOR"      => "00000100",
84    "SUBS"     => "00000101",
85    "ADDS"     => "00000110",
86    "AND"      => "00000111",
87    "LACK"     => "10100001",
88    "LARP"     => "10100011",
89    "LDP"      => "00001001",
90    "LDPK"     => "10100100",
91    "LT"       => "00001010",
92    "LTA"      => "00001011",
93    "LTD"      => "00001100", # 2cycle
94    "LTP"      => "00001101",
95    "LTS"      => "00001110",
```

```
96     "MAR"       => "00001111",
97     "SOVM"      => "00000001",
98     "TBLR"      => "00010001",
99     "TBLW"      => "00010010",
100    "ZALH"      => "00010011",
101    "ZALS"      => "00010100",
102    "SUBSSIMD"=> "00010110",
103    "ADDSSIMD"=> "00010111",
104    "BANZ"      => "00011000",
105    ) ;
106  my %opcode_16bit = (
107    #KEY   => 0123456789ABCDEF
108    "SPAC"=>"0000000100000000",
109    "BU"  =>"0000000100001111",
110    "BGEZ"=>"0000000100000001",
111    "BGZ" =>"0000000100000010",
112    "BLEZ"=>"0000000100000011",
113    "BLZ" =>"0000000100000100",
114    "BNZ" =>"0000000100000101",
115    "BV"  =>"0000000100000110",
116    "BZ"  =>"0000000100000111",
117    "PAC" =>"0000000100011111",
118    "ROVM"=>"0000000100101111",
119    "SOVM"=>"0000000100111111",
120    "NOP" =>"0000000101001111",
121    "ZAC" =>"0000000101011111",
122    "APAC"=>"0000000101101111",
123    "POP" =>"0000000101111111",
124    "PUSH"=>"0000000110001111",
125    "RET" =>"0000000110011111",
126    "CALL"=>"0000000110101111"
127   );
128
129
130  open( my $in_file,'<:encoding(UTF-8)',$assembly_file) or die "\t Error:
       Assembly input file not found!\n";
131  while(<$in_file>){
132          chomp $_;
133          $line_count++;
134          if (($_ =~ /^(.*?);(.*)/) && !($_ =~ /^[\s]*\/\//) ) {
135                  my $code =$1 ;
136                  $code =~ s/^\s+//;
137                  push(@code, $code);
138                  push(@comments, $2);
139                  push(@line_number, $line_count);
140          }
141          elsif (($_ =~ /^[\s]*\/\//)|| ($_ =~ /^[\s]*/)){
142          }
143          else {
```

```
144                     $error_count++;
145                     print "ERROR: Syntax error in line $line_count.";
146             }
147     }
148      ###################################
149      # Machine code generation
150      ###################################
151
152     foreach (my $i=0; $i< @code; $i= $i+1){
153             $code_rearr[$j] = $code[$i];
154             if ($code[$i] =~ /^(.*?)[\s]*:[\s]*(.*)/){
155                     $jmp_in_label_extracted{$1} = $j;
156                     $code[$i] = $2;
157             }
158
159             if ($code[$i] =~ /^(.*?)[\s]+(.*)/){
160                     chomp $1;
161                     chomp $2;
162                     $opcode_extracted[$i] = uc($1);
163                     $non_opcode_extracted[$i] = $2;
164                     if ( exists $opcode_16bit{$opcode_extracted[$i]}){
165                             $opcode_generated[$j] = $opcode_16bit{
                                    $opcode_extracted[$i]};
166                             if ($opcode_extracted[$i] eq "BU"   || #1
167                                     $opcode_extracted[$i] eq "BGEZ"|| #2
168                                     $opcode_extracted[$i] eq "BGZ"  || #3
169                                     $opcode_extracted[$i] eq "BLEZ"|| #4
170                                     $opcode_extracted[$i] eq "BLZ"  || #5
171                                     $opcode_extracted[$i] eq "BNZ"  || #6
172                                     $opcode_extracted[$i] eq "BV"   || #7
173                                     $opcode_extracted[$i] eq "BZ"   || #8
174                                     $opcode_extracted[$i] eq "CALL"){ #9
175
176                                     $j++;
177                                     if ($non_opcode_extracted[$i] =~ /^[\s
                                            ]*(.*?)[\s]*$/i ){
178                                             push (@jmp_labels_called, $1);
179                                             push @{$jmp_out_label_extracted{$1}}
                                                    ,$j;
180                                             # print "Found $1 at $i array @{
                                                    $jmp_out_label_extracted{$1}} \n
                                                    ";
181
182                                             # print "Direct address $1 found;
                                                    machine code= $opcode_generated[
                                                    $i].$non_opcode_generated[$i]
                                                    found in  line".$line_number[$i
                                                    -1]."\n";
183                                     }
```

```
184                                     else {
185                                             $error_count++;
186                                             print "ERROR : Invalid Instruction
                                                    $code[$i] in line ".$line_number[
                                                    $i]."\n";
187                                     }
188
189                             }
190                     }
191 #
    ##########################################################################################

192                     elsif ( exists $opcode_7bit{$opcode_extracted[$i]}){
193                             $opcode_generated[$j] = $opcode_7bit{
                                    $opcode_extracted[$i]};
194                                     if ($non_opcode_extracted[$i] =~ /^0x([0-9A-
                                        F][0-9A-F])[\s]*,[\s]*([GL])[\s]*$/i ){
195                                             $dir_addr = $1;
196                                             $greater_lesser= $2;
197                                             $mode = "0";
198
199                                             if ($greater_lesser eq "G") {
                                                    $greater_lesser = "1";
200                                             }
201                                             else {
                                                    $greater_lesser = "0";
202                                             }
203
204                                             $non_opcode_generated[$j] =
                                                    $greater_lesser.$mode.sprintf ("
                                                    %07b", hex($1));
205                                             # print "Direct address $1 found;
                                                    machine code= $opcode_generated[
                                                    $j].$non_opcode_generated[$j]
                                                    found in  line".$line_number[$i
                                                    -1]."\n";
206                                     }
207                                     elsif ($non_opcode_extracted[$i] =~ /^[\s]*
                                        ar(.*?)[\s]*,[\s]*(.*?)[\s]*,[\s]*([GL])
                                        [\s]*$/i ){
208                                             $incr_oper = $2;
209                                             $ar = $1;
210                                             $greater_lesser= $3;
211                                             $mode = "1";
212
213                                             if ($greater_lesser eq "G") {
                                                    $greater_lesser = "1";
214                                             }
```

```
215                                     else {
                                            $greater_lesser = "0";
216                                     }
217
218                                     if ((exists $add_sub_ar{$incr_oper})
                                            && ($ar <= 7)){
219                                             $non_opcode_generated[$j] =
                                                    $greater_lesser.$mode.
                                                    $add_sub_ar{$incr_oper}."
                                                    00".sprintf ("%03b", $ar)
                                                    ;
220                                     }
221                                     else {
222                                             $error_count++;
223                                             print "ERROR : Invalid
                                                    Instruction $code[$i] in
                                                    line ".$line_number[$i]."
                                                    \n";
224                                     }
225                             }
226                             else {
227                                     $error_count++;
228                                     print "ERROR : Invalid Instruction
                                            $code[$i] in line ".$line_number[
                                            $i]."\n";
229                             }
230                     }
231 #
    ##########################################################################################

232                     elsif ( exists $opcode_8bit{$opcode_extracted[$i]}){
233                             $opcode_generated[$j] = $opcode_8bit{
                                    $opcode_extracted[$i]};
234                             if ($opcode_extracted[$i] eq "LDPK"||
235                                     $opcode_extracted[$i] eq "LACK"||
236                                     $opcode_extracted[$i] eq "MPYK"){
237                                     if ($non_opcode_extracted[$i] =~ /^0x([0-9A-
                                            F][0-9A-F])$/i ){
238                                             $constant = $1;
239                                             $non_opcode_generated[$j] = sprintf
                                                    ("%08b", hex($1));
240                                             # print "constant $1 found; machine
                                                    code= $opcode_generated[$j].
                                                    $non_opcode_generated[$j] found
                                                    in  line".$line_number[$i-1]."\n
                                                    ";
241                                     }
242                                     else {
243                                             $error_count++;
```

```
244                                     print "ERROR : Constant $constant
                                            invalid $code[$i] in line ".
                                            $line_number[$i]."\n";
245                             }
246                     }
247                     elsif($opcode_extracted[$i] eq "LARP"){
248                             if ($non_opcode_extracted[$i] =~ /^[\s]*ar
                                    (.*?)[\s]*$/i ){
249                                     $constant = $1;
250                                     $non_opcode_generated[$j] = sprintf
                                            ("%08b", hex($1));
251                                     # print "constant $1 found; machine
                                            code= $opcode_generated[$j].
                                            $non_opcode_generated[$j] found
                                            in  line".$line_number[$i-1]."\n
                                            ";
252                             }
253                             else {
254                                     $error_count++;
255                                     print "ERROR : Constant $constant
                                            invalid $code[$i] in line ".
                                            $line_number[$i]."\n";
256                             }
257                     }
258                     elsif($opcode_extracted[$i] eq "BANZ"){
259                             $j++;
260                             if ($non_opcode_extracted[$i] =~ /^[\s
                                    ]*(.*?)[\s]*,[\s]*ar(.*?)[\s]*,[\s]*(.*?)
                                    [\s]*$/i ){
261                                     push (@jmp_labels_called, $1);
262                                     push @{$jmp_out_label_extracted{$1}}
                                            ,$j;
263                                     # print "Found $1 at $i array @{
                                            $jmp_out_label_extracted{$1}} \n
                                            ";
264                                     $incr_oper = $3;
265                                     $ar = $2;
266                                     $mode = "1";
267                                     if ((exists $add_sub_ar{$incr_oper})
                                            && ($ar <= 7)){
268                                             $non_opcode_generated[$j-1]
                                                    = $mode.$add_sub_ar{
                                                    $incr_oper}."00".sprintf
                                                    ("%03b", $ar);
269                                             # print "Indirect address
                                                    found; narp= ar$ar &
                                                    operation= $add_sub_ar{
                                                    $incr_oper} machine code=
                                                     $opcode_generated[$j].
```

```
                                        $non_opcode_generated[$j]
                                         found in  line".
                                        $line_number[$i-1]."\n";
270                                 }
271                                 else {
272                                         $error_count++;
273                                         print "ERROR : Invalid
                                            Instruction $code[$i] in
                                            line ".$line_number[$i]."
                                            \n";
274                                 }
275                         }
276                         elsif ($non_opcode_extracted[$i] =~ /^[\s
                            ]*(.*?)[\s]*$/i ){
277                                 push (@jmp_labels_called, $1);
278                                 push @{$jmp_out_label_extracted{$1}}
                                     ,$j;
279                                 # print "Found $1 at $i array @{
                                    $jmp_out_label_extracted{$1}} \n
                                    ";
280
281                                 $mode = "0";
282                                 $non_opcode_generated[$j-1] = $mode.
                                    sprintf ("%07b", hex(0));
283                                 # print "Direct address $1 found;
                                    machine code= $opcode_generated[
                                    $j].$non_opcode_generated[$j]
                                    found in  line".$line_number[$i
                                    -1]."\n";
284                         }
285                         else {
286                                 $error_count++;
287                                 print "ERROR : Invalid Instruction
                                    $code[$i] in line ".$line_number[
                                    $i]."\n";
288                         }
289                 }
290                 else{
291                         if ($non_opcode_extracted[$i] =~ /^0x([0-9A-
                            F][0-9A-F])[\s]*$/i ){
292                                 $dir_addr = $1;
293                                 $mode = "0";
294                                 $non_opcode_generated[$j] = $mode.
                                    sprintf ("%07b", hex($1));
295                                 # print "Direct address $1 found;
                                    machine code= $opcode_generated[
                                    $j].$non_opcode_generated[$j]
                                    found in  line".$line_number[$i
                                    -1]."\n";
```

```
296                                 }
297                                 elsif ($non_opcode_extracted[$i] =~ /^[\s]*
                                        ar(.*?)[\s]*,[\s]*(.*?)[\s]*$/i ){
298                                         $incr_oper = $2;
299                                         $ar = $1;
300                                         $mode = "1";
301                                         if ((exists $add_sub_ar{$incr_oper})
                                            && ($ar <= 7)){
302                                                 $non_opcode_generated[$j] =
                                                    $mode.$add_sub_ar{
                                                    $incr_oper}."00".sprintf
                                                    ("%03b", $ar);
303                                                 # print "Indirect address
                                                    found; narp= ar$ar &
                                                    operation= $add_sub_ar{
                                                    $incr_oper} machine code=
                                                     $opcode_generated[$j].
                                                    $non_opcode_generated[$j]
                                                     found in  line".
                                                    $line_number[$i-1]."\n";
304                                         }
305                                         else {
306                                                 $error_count++;
307                                                 print "ERROR : Invalid
                                                    Instruction $code[$i] in
                                                    line ".$line_number[$i]."
                                                    \n";
308                                         }
309                                 }
310                                 else {
311                                         $error_count++;
312                                         print "ERROR : Invalid Instruction
                                            $code[$i] in line ".$line_number[
                                            $i]."\n";
313                                 }
314                         }
315                 }
316                 elsif ( exists $opcode_5bit{$opcode_extracted[$i]}){
317                         $opcode_generated[$j] = $opcode_5bit{
                                $opcode_extracted[$i]};
318                         if ($opcode_extracted[$i] eq "LARK"){
319                                 if ($non_opcode_extracted[$i] =~ /^[\s]*ar
                                    (.*?)[\s]*,[\s]*0x([0-9A-F][0-9A-F])[\s]*
                                    $/i ){
320                                         $ar = $1;
321                                         $constant = $2;
322                                         if ($ar <= 7){
323                                                 $non_opcode_generated[$j] =
                                                    sprintf ("%03b", $ar).
```

```
                                                    sprintf ("%08b", hex(
                                                    $constant));
324                                                 # print "constant $2 found
                                                    for ar$1 ; machine code=
                                                    $opcode_generated[$j].
                                                    $non_opcode_generated[$j]
                                                     $code[$i] in line ".
                                                    $line_number[$i]."\n";
325                                             }
326                                             else {
327                                                     $error_count++;
328                                                     print "ERROR : Invalid
                                                    Instruction $code[$i] in
                                                    line ".$line_number[$i]."
                                                    \n";
329                                             }
330                                     }
331                             }
332                             else{
333                                     if (($non_opcode_extracted[$i] =~ /^[\s]*ar
                                        (.*?)[\s]*,[\s]*0x([0-9A-F][0-9A-F])[\s]*
                                        $/i )&& ($ar_n <= 7)){
334                                             my $ar_n = $1;
335                                             $dir_addr = $2;
336                                             $mode = "0";
337                                             $non_opcode_generated[$j] = sprintf
                                                ("%03b", hex($ar_n)).$mode.
                                                sprintf ("%07b", hex($2));
338                                             # print "Direct address $1 found;
                                                machine code= $opcode_generated[
                                                $j].$non_opcode_generated[$j]
                                                found in  line".$line_number[$i
                                                -1]."\n";
339                                     }
340                                     elsif ($non_opcode_extracted[$i] =~ /^[\s]*
                                        ar(.*?)[\s]*,[\s]*(.*?)[\s]*,[\s]*ar(.*?)
                                        [\s]*$/i ){
341                                             my $ar_n = $1;
342                                             $incr_oper = $2;
343                                             $ar = $3;
344                                             $mode = "1";
345                                             if ((exists $add_sub_ar{$incr_oper})
                                                && ($ar <= 7)&& ($ar_n <= 7)){
346                                                     $non_opcode_generated[$j] =
                                                        $mode.sprintf ("%03b",
                                                        $ar_n).$add_sub_ar{
                                                        $incr_oper}."00".sprintf
                                                        ("%03b", $ar);
```

```
347                                                 # print "Indirect address
                                                        found; narp= ar$ar &
                                                        operation= $add_sub_ar{
                                                        $incr_oper} machine code=
                                                         $opcode_generated[$j].
                                                        $non_opcode_generated[$j]
                                                         found in  line".
                                                        $line_number[$i-1]."\n";
348                                             }
349                                             else {
350                                                     $error_count++;
351                                                     print "ERROR : Invalid
                                                        Instruction $code[$i] in
                                                        line ".$line_number[$i]."
                                                        \n";
352                                             }
353                                     }
354                                     else {
355                                             $error_count++;
356                                             print "ERROR : Invalid Instruction
                                                $code[$i] in line ".$line_number[
                                                $i]."\n";
357                                     }
358                             }
359                     }
360                     elsif ( exists $opcode_3bit{$opcode_extracted[$i]}){
361                             $opcode_generated[$j] = $opcode_3bit{
                                $opcode_extracted[$i]};
362                             if ($non_opcode_extracted[$i] =~ /^0x([0-9A-F][0-9A-
                                F])[\s]*,[\s]*(\d+)[\s]*$/i ){
363                                     $dir_addr = $1;
364                                     $mode = "0";
365                                     $shift = $2;
366                                     $non_opcode_generated[$j] = sprintf ("%05b",
                                          $shift).$mode.sprintf ("%07b", hex($1));
367                                     # print "Direct address $1 found; machine
                                        code= $opcode_generated[$j].
                                        $non_opcode_generated[$j] found in  line
                                        ".$line_number[$i-1]."\n";
368                             }
369                             elsif ($non_opcode_extracted[$i] =~ /^[\s]*ar(.*?)[\
                                s]*,[\s]*(.*?)[\s]*,[\s]*(.*?)[\s]*$/i ){
370                                     $incr_oper = $2;
371                                     $shift = $3;
372                                     $ar = $1;
373                                     $mode = "1";
374                                     if ((exists $add_sub_ar{$incr_oper})&& ($ar
                                        <= 7)){
```

```
375                                         $non_opcode_generated[$j] = sprintf
                                                ("%05b", $shift).$mode.
                                                $add_sub_ar{$incr_oper}."00".
                                                sprintf ("%03b", $ar);
376                                         # print "Indirect address found;
                                                narp= ar$ar & operation=
                                                $add_sub_ar{$incr_oper} machine
                                                code= $opcode_generated[$j].
                                                $non_opcode_generated[$j] found
                                                in  line ".$line_number[$i-1]."\n
                                                ";
377                                     }
378                                     else {
379                                         $error_count++;
380                                         print "ERROR : Invalid Instruction
                                                $code[$i] in line ".$line_number[
                                                $i]."\n";
381                                     }
382                                 }
383                                 else {
384                                     $error_count++;
385                                     print "ERROR : Invalid Instruction $code[$i]
                                            in line ".$line_number[$i]."\n";
386                                 }
387
388                         }
389                         else{
390                             $error_count++;
391                             print "ERROR : Opcode doesn't exist $code[$i] in
                                    line ".$line_number[$i]."\n";
392                         }
393                         # print $opcode_extracted[$i]."\t".
                                $non_opcode_extracted[$i]."\t;".$opcode_generated
                                [$j]."\t".$non_opcode_generated[$j]."\n";
394                         $j++;
395                 }
396                 elsif (exists $opcode_16bit{uc($code[$i])}){
397                     $opcode_generated[$j] = $opcode_16bit{uc($code[$i])};
398                     $j++;
399                 }
400                 else{
401                     $error_count++;
402                     print "ERROR : Instruction $code[$i] in line ".$line_number[$i]."
                            DOES NOT EXIST\n";
403                 }
404 }
405  ###################################
406  # Jump address generation
407  ###################################
```

```
408
409 foreach my $k (@jmp_labels_called){
410     if (exists $jmp_in_label_extracted{$k}){
411                 my @temp_arr = @{$jmp_out_label_extracted{$k}};
412                 # print "Label $k found in: $jmp_in_label_extracted{$k}\n";
413                 foreach my $jmp_addr (@temp_arr){
414                     $jmp_addr_generated[$jmp_addr] = sprintf ("%016b",
                            $jmp_in_label_extracted{$k});
415             $jmp_addr_generated_nonconverted[$jmp_addr] = "JA =".
                    $jmp_in_label_extracted{$k};
416                 }
417         }
418 }
419
420 if ($error_count == 0){
421 #
    ##########################################################################################

422         open( my $out_file,'>',$mif_file) or die "\t Error: Output MIF file
                not found and can't be created!\n";
423         print $out_file "WIDTH = 16;\nDEPTH = 65536;\n\nADDRESS_RADIX = DEC;
                % Can be HEX, BIN or DEC %\nDATA_RADIX = BIN;    % Can be HEX,
             BIN or DEC%\n\n\nCONTENT BEGIN\n";
424
425         for (my $i =0 ; $i < $j; $i++){
426                 my $int = $opcode_generated[$i].$non_opcode_generated[$i].
                        $jmp_addr_generated[$i];
427                 $int = unpack("N", pack("B32", substr("0" x 32 .$int, -32)))
                        ;
428                 my $hex = sprintf("%04x", $int );
429                 print $out_file $i.":".$hex.";%\t\t".$code_rearr[$i].
                        $jmp_addr_generated_nonconverted[$i]."%\n";
430 #               print $out_file $i.":".$opcode_generated[$i].
    $non_opcode_generated[$i].$jmp_addr_generated[$i].";%\t\t".$code_rearr[$i
    ].$jmp_addr_generated_nonconverted[$i]."%\n";
431         }
432         print $out_file "END;";
433
434         close($out_file);
435         print"\nASSEMBLY SUCCESSFUL : Successfully assembled $assembly_file
                .\n\tPlease check $mif_file for the output.\n" ;
436 #
    ##########################################################################################

437         open( my $hex_file_out,'>',$hex_file) or die "\t Error: Output HEX
                file not found and can't be created!\n";
438
439         for (my $i =0 ; $i < $j; $i++){
```

```
440                 my $int = $opcode_generated[$i].$non_opcode_generated[$i].
                        $jmp_addr_generated[$i];
441                 $int = unpack("N", pack("B32", substr("0" x 32 .$int, -32)))
                        ;
442                 my $hex = sprintf("%04x", $int );
443                 print $hex_file_out $hex."\n";
444         }
445
446         close($hex_file_out);
447         print"\n\tPlease check $hex_file for the Hex output.\n" ;
448 #
    ##########################################################################################

449 }
450 else {
451         print"\nASSEMBLY FAILED : Total of $error_count errors found. Please
              check your code\n" ;
452 }
```

I.3 Assembly source code for testing and median filter

I.3.1 Assembly code used for basic level testing

```
//
   ////////////////////////////////////////////////////////////////////////

////////// Version 2  //////////
//
   ////////////////////////////////////////////////////////////////////////

//
   ////////////////////////////////////////////////////////////////////////

lack 0x01;          //  acc= 1  arp=x    ar0=0    ar1=0
sac 0x00, 0;        //  acc= 1  arp=x    ar0=0    ar1=0   mem[0]=1
lack 0x03;          //  acc= 3  arp=x    ar0=0    ar1=0   mem[0]=1
lark ar0, 0x00;     //  acc= 3  arp=0    ar0=0    ar1=0   mem[0]=1
add *+,0,ar0;       //  acc= 4  arp=0    ar0=1    ar1=0   mem[0]=1
sac *-,0,ar0;       //  acc= 4  arp=0    ar0=0    ar1=0   mem[0]=1
   mem[1]=4
sub *+,0,ar1;       //  acc= 3  arp=0    ar0=0    ar1=0   mem[0]=1
   mem[1]=4
lark ar1, 0x10;     //  acc= 3  arp=1    ar0=1    ar1=10  mem[0]=1
   mem[1]=4
```

```
sac *,0,ar0;        // acc= 3 arp=1 ar0=1 ar1=10 mem[0]=1,
   mem[1]=4,mem[10]=3
lt *,ar1;           // acc= 3 arp=0 ar0=1 ar1=10 mem[0]=1,
   mem[1]=4,mem[10]=3     ; Treg=4
mpy *, ar0;         // acc= 3 arp=1 ar0=1 ar1=10 mem[0]=1,
   mem[1]=4,mem[10]=3     ; Treg=4 Preg=C
mac *-, ar0;        // acc= f arp=0 ar0=1 ar1=10 mem[0]=1,
   mem[1]=4,mem[10]=3     ; Treg=4 Preg=f (acc=c+acc;Preg=4*mem[
   ar0])
mpyk 0x02;          // acc= f arp=0 ar0=1 ar1=10 mem[0]=1,
   mem[1]=4,mem[10]=3     ; Treg=4 Preg=8
pac ;
L1: sub *,0,ar0;
bnz L1;
bz L2;
L3: or *, ar1;
xor *, ar1;
ret;
L2: call L3;
nop;
//
   ////////////////////////////////////////////////////////////////
```

I.3.2 Assembly code used for median filter algorithm

```
LAR AR0, 0x0003;
LAR AR1, 0x0004;
LAR AR2, 0x0005;
LAR AR3, 0x0006;
LAR AR4, 0x0007;
LAR AR5, 0x0008;
LAR AR7, 0x0009;
LARP AR7;
LAC 0x0000, 0;
SAC AR7, *-,0;
LACK 0x01;
SAC AR7, *+,0;
MAR AR7, *+;
LAC 0x0001, 0;
SAC AR7, *+,0;
LACK 0x01;
SAC AR7,*-,0;
MAR AR0, *-;
L1: LAC AR0, *+, 0;
CMPSIMD AR0,*- ,G;
PUSH;
PUSH;
LAC AR0, *+, 0;
```

```
CMPSIMD AR0,*+  ,L;
PUSH;
CMPSIMD AR3,*-,G;
SAC AR0, *, 0;
POP;
CMPSIMD AR5,*,L;
SAC AR3, *-, 0;
POP;
CMPSIMD AR3,*-,G;
SAC AR3, *+, 0;
POP;
CMPSIMD AR3,*,L;
SAC AR1, *+, 0;
LAC AR1, *+, 0;
CMPSIMD AR1,*-  ,G;
PUSH;
PUSH;
LAC AR1, *+, 0;
CMPSIMD AR1,*+  ,L;
PUSH;
CMPSIMD AR4,*-,G;
SAC AR1, *, 0;
POP;
CMPSIMD AR3,*,L;
SAC AR4, *, 0;
```

```
POP;
CMPSIMD AR4,*-,G;
SAC AR4, *+, 0;
POP;
CMPSIMD AR4,*,L;
SAC AR2, *+, 0;
LAC AR2, *+, 0;
CMPSIMD AR2,*- ,G;
PUSH;
PUSH;
LAC AR2, *+, 0;
CMPSIMD AR2,*+ ,L;
PUSH;
CMPSIMD AR4,*-,G;
SAC AR2, *, 0;
POP;
CMPSIMD AR5,*,L;
SAC AR4, *-, 0;
POP;
CMPSIMD AR5,*,G;
SAC AR4, *+, 0;
POP;
CMPSIMD AR4,*,L;
SAC AR3, *-, 0;
LAC AR5, *-, 0;
```

```
CMPSIMD AR5,*+  ,G;
CMPSIMD AR5,*-  ,G;
SAC AR3, *-, 0;
LAC AR4, *, 0;
CMPSIMD AR3,*  ,G;
PUSH;
LAC AR4, *-, 0;
CMPSIMD AR4,*+,L;
CMPSIMD AR4,*-,G;
SAC AR4 *, 0;
POP;
CMPSIMD AR4,*,L;
SAC AR3, *-, 0;
LAC AR4, *+, 0;
CMPSIMD AR5,*+,L;
CMPSIMD AR3,*+,L;
SAC AR3, *, 0;
LAC AR4, *, 0;
CMPSIMD AR3,*,G;
PUSH;
LAC AR4, *, 0;
CMPSIMD AR5,*,L;
CMPSIMD AR4,*,G;
SAC AR4 *, 0;
POP;
```

```
CMPSIMD AR6,*,L;
SAC AR7, *+, 0;
LAC AR7, *-,0;
SUB AR7, *+,0;
SAC AR0, *,0;
BNZ L1;
MAR AR0, *+,0;
MAR AR1, *+,0;
MAR AR1, *+,0;
MAR AR2, *+,0;
MAR AR2, *+,0;
MAR AR7,*+,0;
LAR AR7, 0x0000;
MAR AR7, *+;
LAC AR7, *+,0;
SUB AR7, *-,0;
SAC AR0, *,0;
BNZ L1;
L2:LACK 0x00;
SAC AR6,*+,0;
BU L2;
```